SF 298 MASTER COPY   KEEP THIS COPY FOR REPRODUCTION PURPOSES

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB NO. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE November 24, 1997	3. REPORT TYPE AND DATES COVERED Final Progress Report, 1993-1997

4. TITLE AND SUBTITLE The Mark2 Theorem Prover	5. FUNDING NUMBERS DAAH04-94-G-0247
6. AUTHOR(S) M. Randall Holmes	
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES) Boise State University Boise, ID 83725	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S. Army Research Office P.O. Box 12211 Research Triangle Park, NC 27709-2211	10. SPONSORING / MONITORING AGENCY REPORT NUMBER ARO 33580.5-MA-DPS

11. SUPPLEMENTARY NOTES

The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12 b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Mark2, a general-purpose system for computer-aided reasoning, is described. Mark2 is an equational system of reasoning implementing a higher-order logic somewhat stronger than simple type theory. It incorporates a tactic language in which tactics are represented as a kind of equational theorem, proved in much the same way as ordinary theorems. Reasoning with expressions defined by cases is a focus of the Mark2 logic. The higher order logic of Mark2 is unusual in being untyped; it is a safe version of Quine's "New Foundations", implemented as a λ-calculus. Documentation for the prover is attached as an appendix.

19971215 086

14. SUBJECT TERMS	15. NUMBER IF PAGES 80 + 94
	16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

The Mark2 Theorem Prover*

M. Randall Holmes

November 25, 1997

Contents

*supported by ARO grant DAAH04-94-0247

1 Introduction

This paper will describe Mark2, a general-purpose system for computer aided reasoning developed by the author. In its current version, it is the final progress report for ARO grant DAAH04-94-0247. The support of the Army Research Office is appreciated.

The current version of the prover documentation is attached as an appendix. The prover source code and a collection of proof scripts can be obtained from our Web page, `http://math.idbsu.edu/~holmes` (follow the link to Mark2).

The prover implements a higher order logic somewhat stronger than the usual simple theory of types. This logic is implemented in three layers.

The first layer is devoted to equational or algebraic reasoning: all Mark2 theories are equational in the sense that their theorems are equations and are used as rewrite rules. The tactic language of Mark2 is based on a device for expressing recursively chained rewrite rules as theorems within a Mark2 theory; the basic ideas of the tactic language live in the first layer, though it acquires refinements as we consider the second and third layers.

The second layer is devoted to reasoning about expressions defined by cases. This layer of the logic implements propositional logic and the logic of identity.

The third layer is devoted to abstraction. Quantification and a strong higher order logic are implemented on this level. The higher order logic used is unusual in being (at least superficially) type-free; it is a version of Quine's set theory "New Foundations" (see [21], [18]).

The three layers of the logic provide an organizational principle for this paper; each of the layers is considered in turn.

There are many examples of prover sessions in the text; they are not as a rule realistic examples of applications, being generally very low-level in their approach. More efficient proofs become possible when more groundwork has been done than is possible in the scope of a surveyable example.

2 The Algebraic Layer and the Tactic Language

This section discusses the features of Mark2 which support equational reasoning and the implementation of tactics (programs which automatically carry out many proof steps) as "equational theorems" of a special form.

2.1 The Term Language of Mark2

In this section, we summarize those features of the term language of Mark2 which are relevant at the algebraic level.

Any string which contains no characters other than letters, digits or the special characters ? and _ may be an atomic term of the language of Mark2.

Atomic terms of the language of Mark2 are of four kinds:

numerals: Strings of digits are recognized by Mark2 as numerals. Unlike other atomic constants, numerals do not need to be declared. Mark2 provides built-in operations of infinite-precision unsigned integer arithmetic as part of the tactic language.

constants: An atomic term which contains a non-digit and does not begin with ? is a *constant*. Constants used in a given Mark2 theory must be declared in that theory.

bound variables: An atomic term which consists of ? followed by a non-zero-initial numeral is a *bound variable*. The function of bound variables is discussed in the section on the "abstraction layer".

free variables: An atomic term which begins with ? and is not a bound variable is a *free variable*. Any free variable may be used in any theory without declaration.

A string of special characters not including ?, _ or paired forms like parentheses, braces, and brackets is called an *operator*.

An operator of more than one character beginning with ^ is an *operator variable*. Operator variables can be used without declaration, though there are situations in which it is desirable to declare operator variables as having special features.

An operator of more than one character beginning with : is a *type-raised operator*, the result of one or more applications of a certain operation to the operator (variable or constant) obtained by stripping off the initial occurrences of :. This operation is discussed in the section on the "abstraction layer".

All other operators (including the one character operators : and ^) are *operator constants*. Operator constants must be declared in a theory in which they are used; there are a number of operators which are automatically declared by Mark2.

Complex terms of Mark2 are of three kinds:

infix terms: These consist of a term followed by an operator followed by a term. Parentheses may be required, depending on the precedence of operators involved. Parentheses may be used freely in terms to be parsed by the prover; the prover will display only those parentheses which are needed. Precedence of operators is set by the user; the default is for all operators to have the same precedence and group to the right (the convention of APL).

prefix terms: These consist of an operator followed by a term. Remarks about parentheses are the same as in the case of infix terms. A prefix term is always regarded by Mark2 as being an abbreviation for an infix term; an operator cannot be used as a prefix unless a declaration has been made of the "default left argument" to be understood as the left argument of the infix terms of which terms with the operator as prefix are abbreviations.

bracket terms: These consist of a term enclosed in brackets. If a term contains no bound variables, the result of enclosing it in brackets is the constant function with the referent of the enclosed term as its constant value. The general function of brackets is to define functions by abstraction: bracket terms are λ-terms. DeBruijn levels (see [5] for the origin of this concept) are used to determine variable binding: outermost bracket subterms of a term bind `?1`, all second-to-outermost bracket subterms bind `?2` and so forth. A bound variable may not appear in a context in which it is not bound. This will be discussed further in the section on the "abstraction layer".

Syntactical refinements which are expected to be added are postfix terms and more general forms for operators (allowing alphanumeric characters to be used in names of operators).

2.2 Substitution and Rewriting

For the moment, we restrict ourselves to the subset of the language without bound variables. Bracketed terms are present, with `[T]` denoting the constant function whose value everywhere is `T`. The virtue of this subset of the language is that the definition of substitution is very simple. If `T` is a term, we use the notation `T[A/?x]` to denote the result of replacing the free variable `?x` with the term `A` wherever the variable appears in `T`.

The set of *substitution instances* of a term `T` is the smallest set of terms which contains `T` itself and contains the term `U[A/?x]` for each term `A` and free variable `?x` whenever it contains `U`.

A term of the form `A = B` is called an *equation* (unsurprisingly). We say that a term `T` is rewritten to the term `U` by the equation `A = B` whenever the equation `T = U` is a substitution instance of the equation `A = B`. This has one peculiar consequence: if `B` happens to contain free variables which do not occur in `A`, then there are many possible results of rewriting `T` with `A = B`. In practice, variables in `B` which do not occur in `A` are rewritten to new variables (i.e., variables wich did not occur in `A`).

If `A = B` is an equation, the equation `B = A` is said to be the converse of `A = B`. Equations `A = B` and `B = C` are said to have composition `A = C` (composition can be given a somewhat more general definition, but this suffices for our purposes).

If `A = B` is an equation, we define **R**(`A = B`) as the equation `(?new ^+ A) = (?new ^+ B)`, where the variable `?new` and the infix variable `^+` do not appear in A or B (strictly speaking, this does not define a unique equation, but the notion of rewriting with all such equations is the same). Similarly, we define **L**(A = B) as (A ^+ ?new) = (B ^+ ?new) and **V**(A = B) as [A] = [B]. We refer to **R**, **L**, and **V** as "localizing operations" on equations. We formalize the notion of applying a finite sequence of localizing operations to an equation: the empty sequence applied to an equation gives the equation itself, the application of a one-term sequence (O_1) is the same as the application of O_1, while the application of the sequence $(O_1, \ldots, O_n)$ of localization operators O_i to an equation A = B is the result of applying $(O_1, \ldots, O_{n-1})$ to the equation O_n(A = B).

A Mark2 "abstract theory" is a set of equations containing all identity equations T = T (mod considerations of declared constants and operations) and closed under converse, composition, substitution instance, and localizing operations. It should be clear that the notion of a Mark2 abstract theory coincides with the notion of a set of equations closed under the usual methods of proof in equational theories (within the bounds of the details of Mark2 notation). A "concrete theory" in Mark2 is simply a set of equations (regarded as representing the minimal abstract theory of which it is a subset).

2.3 The Basic Session Model

The aim of a session under Mark2 is the proof of an equational theorem in a given theory. One is given a set of theorems already postulated or proved (a concrete theory); the aim is to demonstrate that a desired equation can be added to the concrete theory, i.e., that it can be obtained from the sentences in the concrete theory by a combination of the operations on equations listed in the previous section.

Initially the user enters a term T. The user then sees the term T, but the object actually being manipulated is the equation T = T. A sequence of transformations are applied to this equation, each of which is guaranteed to preserve the condition that the equation produced will belong to the minimal abstract theory containing the given concrete theory. The allowed transformations are the conversion of the equation (interchange of left and right hand sides), global assignment of a value to a free variable (repeated application of this justifies replacement of the equation by any substitution instance) and rewriting of the right hand side of the equation by an element of the theory or by the result of application of a sequence of localizing operations to an element of the theory.

At each step, the prover is manipulating an equation internally, but the user sees things rather differently. What the user is presented with is a term, the right hand side of the equation which the prover is considering, and a selected subterm of that term. The choice of selected subterm encodes the sequence of localizing operations which will be applied to any equation of the concrete theory invoked as a rewrite rule. In other words, the selected subterm is the specific subterm which an equation from the concrete theory may be used to rewrite. The user may manipulate the selection of this subterm by "moving" through the term, either to the left (an **L** or **V**, depending on the form of the previous selected subterm is added to the sequence of these operations to be applied), to the right (precisely analogous to movement to the left), up (the sequence of localizing operations to be applied has its last element deleted, so rewriting will be carried out on the "parent" term of the previously selected subterm) or to the top (the list of localizing operations is emptied and rewrites will be applied to the entire right-hand side of the equation). More sophisticated movement commands are available, but this is the basic set. The sequence of localizing operations is actually represented by a list of booleans (truth values), in which `true` represents **L**, `false` represents **R**, and either truth value may indifferently represent `V`.

The options available from the user's standpoint are to interchange the left and right hand sides of the equation (thus replacing the right-hand term with the left-hand term as the term being viewed), make a global assignment to a free variable (this may affect both sides of the equation being proved), navigate within the term he/she sees (change the selection of the subterm to which rewrites are to be applied), or apply equations in the concrete theory as rewrite rules to the selected subterm of the right hand side of the equation. When the equation has achieved the desired form, it may be added to the concrete theory; an identifier is declared to represent the newly proved theorem.

This is a model of equational proof which is adequate in theory but of course tedious in practice. It differs from the usual approach to proof with rewrite rules in requiring complete control over where rules are applied (the usual approach can be realized using the tactic language). The introduction of the metaphor of navigation through the parse tree of the term being manipulated goes some way toward making this sort of reasoning feasible.

2.4 The Tactic Language

One of the novel features of Mark2 is the ability to represent "tactics" (programs for the automatic execution of many proof steps) as equational theorems, stored in theories in the same way as conventional theorems. The term "tactic" is borrowed from the provers of the LCF family such as Nuprl ([3]) or HOL ([8]), but in those systems a tactic is an ML function, an object of quite a different sort than a theorem.

Mark2 provides predeclared operators which allow one to represent in a term, without changing its semantics, the intention of rewriting with an equation. A term of the form `thm => term` has the same referent as `term`, but has the additional connotation that it is intended to apply the equation in the current theory with the name `thm`. It is thus required that theorems in the current theory have names which are declared constants in the current theory. A term `thm <= term` has a similar additional connotation, except that the converse of the theorem referred to by `thm` is to be applied.

Special commands allow the introduction of such rewriting annotations. When the tactic interpreter is invoked, the intentions signalled by all such rewriting annotations are carried out. When the theorem `thm` cannot be used to rewrite the term `thm => term` or `thm <= term` in the indicated sense, the annotation is simply dropped. The tactic interpreter follows a depth-first strategy; the term `thm => term` or `thm <= term` is not considered for rewriting until `term` itself is reduced to a form free of annotations.

The power of this idea is seen when it is realized that one can prove equations as theorems which involve rewriting annotations. Rewriting with such theorems will introduce new rewriting annotations; the tactic interpreter carries these out in the same way that it carries out annotations present when it is invoked.

The tactic language of Mark2 is also discussed in our [14].

2.5 An Example Session

The prover is implemented in Standard ML, and its interface is currently the SML interface. We do not claim that this is a desirable state of affairs; a better interface is one of our goals.

In this subsection, we will present examples of declarations setting up a small theory, and some simple proof sessions, in order to give a concrete referent to the discussion up to this point.

In all sessions presented in this paper, it is useful to be aware that the SML prompt is a hyphen `-`; this allows one to distinguish commands given by the user to the prover from prover responses.

We first declare an operator.

```
- declareinfix "+";
```

At this point, the prover knows nothing about this operator: we continue by supplying some basic axioms.

```
- axiom "COMM" "?x+?y" "?y+?x";

COMM:
?x + ?y =
?y + ?x
["COMM"]

- axiom "ASSOC" "(?x+?y)+?z" "?x+?y+?z";

ASSOC:
(?x + ?y) + ?z =
?x + ?y + ?z
["ASSOC"]

- axiom "ZERO" "0+?x" "?x";

ZERO:
0 + ?x =
?x
["ZERO"]
```

The command and the prover's responses are given. In connection with the associativity axiom, it is worth recalling here that the default grouping of all operators is to the right.

We first prove a simple theorem.

```
- start "?x+?y+?z";

{?x + ?y + ?z}

?x + ?y + ?z
```

We enter the term which will serve as the left side of the theorem to be proved.

```
- right();

?x + {?y + ?z}

?y + ?z
```

We "move" to the right subterm.

```
- ruleintro "COMM";

?x + {COMM => ?y + ?z}

COMM => ?y + ?z
```

We indicate the intention of rewriting this subterm with the commutative law.

```
- execute();

?x + {?z + ?y}

?z + ?y
```

We carry out this intention: `execute` is the command which invokes the tactic interpreter.

```
- up();

{?x + ?z + ?y}

?x + ?z + ?y

- ruleintro "COMM"; execute();

{COMM => ?x + ?z + ?y}

COMM => ?x + ?z + ?y
```

```
{(?z + ?y) + ?x}

(?z + ?y) + ?x
```

We move back up to the top of the term and apply the commutative law.

```
- ruleintro "ASSOC"; execute();

{ASSOC => (?z + ?y) + ?x}

ASSOC => (?z + ?y) + ?x

{?z + ?y + ?x}

?z + ?y + ?x
```

We apply the associative law. This completes the proof of the intended theorem; we now record the theorem as proved.

```
- prove "ROTATE";

ROTATE:
?x + ?y + ?z =
?z + ?y + ?x
["ASSOC","COMM"]
```

The new theorem is assigned the name ROTATE. It can be invoked in the same way the axioms were invoked:

```
- start "?a+?b+?c+?d";

{?a + ?b + ?c + ?d}

?a + ?b + ?c + ?d

- ri "ROTATE"; execute();

{ROTATE => ?a + ?b + ?c + ?d}

ROTATE => ?a + ?b + ?c + ?d

{(?c + ?d) + ?b + ?a}

(?c + ?d) + ?b + ?a

- ri "ROTATE"; execute();
```

```
{ROTATE => (?c + ?d) + ?b + ?a}

ROTATE => (?c + ?d) + ?b + ?a

{?a + ?b + ?c + ?d}

?a + ?b + ?c + ?d
```

Now we show how simple tactics are developed. After this point in the paper, we will not show displays of the selected subterm when it is at the top level.

```
- s "?x+0";

{?x + 0}

- ri "COMM"; execute();

{COMM => ?x + 0}

{0 + ?x}

- ri "ZERO"; execute();

{ZERO => 0 + ?x}

{?x}

- prove "COMMZERO";

COMMZERO:
?x + 0 =
?x
["COMM","ZERO"]
```

The theorem COMMZERO is an ordinary equational theorem, of course.

```
- start "?x+?y";

{?x + ?y}

- ruleintro "ZERO"; ruleintro "COMMZERO";

{ZERO => ?x + ?y}

{COMMZERO => ZERO => ?x + ?y}
```

```
- prove "EITHERZERO";

EITHERZERO:
?x + ?y =
COMMZERO => ZERO => ?x + ?y
["COMM","ZERO"]
```

The new "theorem" EITHERZERO involves rewriting annotations; it is a simple tactic.

```
- start "?x+0";

{?x + 0}

- ruleintro "EITHERZERO"; execute();

{EITHERZERO => ?x + 0}

{?x}

- start "0+?x";

{0 + ?x}

- ruleintro "EITHERZERO"; execute();

{EITHERZERO => 0 + ?x}

{?x}
```

We see that EITHERZERO has an effect which cannot be achieved by rewriting with a single equation; it eliminates addition of zero on the right or the left.

```
- declarepretheorem "ZEROES";

- start "0+?x";

{0 + ?x}

- ri "ZERO"; execute();

{ZERO => 0 + ?x}

{?x}

- ri "ZEROES";
```

```
{ZEROES => ?x}

- prove "ZEROES";

ZEROES:
0 + ?x =
ZEROES => ?x
["ZERO"]
```

The special declaration with `declarepretheorem` is required because the `ruleintro` command checks for declarations of theorems introduced; by the nature of the situation, ZEROES must be introduced before it has actually been proved (and so declared). The effect of rewriting with ZEROES is more impressive than that of rewriting with EITHERZERO:

```
- start "0+0+0+0+0+0+?x";

{0 + 0 + 0 + 0 + 0 + 0 + ?x}

- ri "ZEROES"; execute();

{ZEROES => 0 + 0 + 0 + 0 + 0 + 0 + ?x}

{?x}
```

The effect of ZEROES is to eliminate addition of zero on the left as many times as it can.

```
- startover();

{0 + 0 + 0 + 0 + 0 + 0 + ?x}

- ri "ZEROES"; steps();

{ZEROES => 0 + 0 + 0 + 0 + 0 + 0 + ?x}

ZEROES => 0 + 0 + 0 + 0 + 0 + 0 + ?x
ZEROES => 0 + 0 + 0 + 0 + 0 + ?x
ZEROES => 0 + 0 + 0 + 0 + ?x
ZEROES => 0 + 0 + 0 + ?x
ZEROES => 0 + 0 + ?x
ZEROES => 0 + ?x
ZEROES => ?x
?x
```

We recapitulate the last proof. The `steps` command invokes the "tactic debugger", which traces the effect of `execute` step by step. Note that it is important that the rewriting annotation with ZEROES will be dropped when it no longer applies; this is why it is possible for the recursion to terminate. These examples, while very simple, should suggest why the tactic interpreter of Mark2 supports, in effect, a full-fledged programming language. Refinements described in the next section increase the ease of use of this programming language.

2.6 Refinements of the Tactic Language

In this section, some refinements of the tactic language of Mark2 are discussed, with motivating examples.

The first refinement is best illustrated by considering the tactic EITHERZERO developed in the previous subsection. It turns out that this tactic has unexpected behavior in certain circumstances:

```
- s "0+?x+0";

{0 + ?x + 0}

- ri "EITHERZERO"; execute();

{EITHERZERO => 0 + ?x + 0}

{?x}
```

The difficulty here is that we may have thought of the applications of ZERO and COMMZERO as alternatives, but there is a case where one can be applied and then the other.

The solution is the definition of another pair of infixes for rewriting annotation: the new infixes =>> and <<= signal the intention of applying an equation as a rewrite rule if a previous attempt to rewrite has failed. In an expression of the form `thm_n =>>...thm_2 =>> thm_1 => term`, the tactic interpreter will attempt to apply each of the theorems in turn, starting with `thm_1`; once a theorem applies, the subsequent theorems (working outward) will be ignored. The innermost annotation infix must be => (or <=) because there is no previous rewrite in the sequence which might fail.

```
- s "?x+?y";

{?x + ?y}

- ruleintro "ZERO";
{ZERO => ?x + ?y}

- altruleintro "COMMZERO";

{COMMZERO =>> ZERO => ?x + ?y}

- prove "EITHERZERO2";
```

```
EITHERZERO2:
?x + ?y =
COMMZERO =>> ZERO => ?x + ?y
["COMM","ZERO"]

- s "0+?x+0";

{0 + ?x + 0}

- ri "EITHERZERO2";

{EITHERZERO2 => 0 + ?x + 0}

- execute();

{?x + 0}
```

Here we see more natural behavior for EITHERZERO2; it eliminates one addition of zero on either the right or the left. This is a trivial example, of course; it has turned out to be important in ensuring reliable behavior of more complex tactics to be able to be certain that only one of a list of alternatives would be applied.

The second major refinement of the tactic language allows parameters to be supplied to tactics. The infix @ is the predeclared infix of function application; it is also used to link parameters to tactics (parameterized tactics can be thought of as theorem-valued functions). Multiple parameters can be linked with the predeclared pair infix ,. Here is a simple example of a parameterized tactic:

```
- start "?x^+?y";

{?x ^+ ?y}

- right();

?x ^+ {?y}

?y

- ruleintro "?thm";

?x ^+ {?thm => ?y}

?thm => ?y

- prove "RIGHT@?thm";
```

```
RIGHT @ ?thm:
?x ^+ ?y =
?x ^+ ?thm => ?y
[]

- start "?x+?y+?z";

{?x + ?y + ?z}

- ruleintro "RIGHT@COMM"; execute();

{(RIGHT @ COMM) => ?x + ?y + ?z}

{?x + ?z + ?y}
```

The parameterized tactic RIGHT allows one to apply a theorem (supplied as the parameter) to the right subterm of the target term.

Parameterized tactics can have operators as names:

```
- start "?x";

{?x}

- ruleintro "?thm1"; ruleintro "?thm2";

{?thm1 => ?x}

{?thm2 => ?thm1 => ?x}

- prove "?thm1**?thm2";

?thm1 ** ?thm2:
?x =
?thm2 => ?thm1 => ?x
[]
```

The infix ** has been "proved" as a theorem implementing theorem or tactic composition.

Another elementary use of parameterization is to control the effects of the converses of theorems which "destroy information":

```
- thmdisplay "REFLEX";

REFLEX:
?a = ?a =
true
["REFLEX"]
```

```
- start "true";

{true}

- revruleintro "REFLEX"; execute();

{REFLEX <= true}

{?a_10 = ?a_10}
```

Note that the converse of REFLEX introduces a computer-generated new variable. We would like to have control over what is introduced.

```
- start "true";

{true}

- initializecounter();

- revruleintro "REFLEX"; execute();

{REFLEX <= true}

{?a_1 = ?a_1}

- assign "?a_1" "?a";

{?a = ?a}

- prove "REV_REFLEX@?a";

REV_REFLEX @ ?a:
true =
?a = ?a
["REFLEX"]

- start "true";

{true}

- ruleintro "REV_REFLEX@3"; execute();

{(REV_REFLEX @ 3) => true}

{3 = 3}
```

The `initializecounter` command is a technicality; it forces the counter on computer-generated new variables back to 1 so that the proof will work correctly if run as a script (there would be problems with the `assign` command otherwise). One can see examples here of the use of converses of theorems as rewrite rules and of the use of global assignment. One can also see that a parameterized tactic can take objects of a theory as parameters as well as theorems or tactics.

2.7 An Extended Example: Combinatory Logic

As an extended example of the capabilities of Mark2 as a rewriting system, we present a development of untyped combinatory logic in the style of Curry (see [4]).

The theory *CL* is an equational theory. Objects of the theory are called "combinators". Atomic terms of the language of this theory are atomic constants I, S, K, B, C, and any others that may be desired, plus an infinite supply of variables. The only term construction is function application: if T and U are terms, (TU) is a term. Excess parentheses are deleted from the left (the opposite of the default Mark2 convention): $ABCD$ is read $(((AB)C)D)$.

The axioms of *CL* are as follows:

II: $IX = X$ (for any term X)

KK: $KXY = X$ (for any terms X and Y)

SS: $SXYZ = XZ(YZ)$ (for any terms X, Y, and Z)

Axioms could be provided for the combinators B and C (of composition and conversion), but it is more natural to introduce these combinators by definition (as we will see in the development below).

We begin with declarations and axioms:

```
- declareinfix ".";  (* this is the "function application" internal to CL *)
- setprecedence "." 1;  (* this gives the correct grouping *)
- declareconstant "I";  (* atomic combinators *)
- declareconstant "K";
- declareconstant "S";
- axiom "II" "I . ?x" "?x";
- axiom "KK" "K . ?x . ?y" "?x";
- axiom "SS" "S . ?x . ?y . ?z" "?x . ?z . (?y . ?z)";
(* the output from these axiom declarations is omitted *)
```

The `setprecedence` command is used to tell the prover that the . operator of "function application" (in the *CL* sense) groups to the left rather than to the right. Mark2 regards all operators with odd precedence as grouping to the right and all operators with even precedence as grouping to the right.

In this section, some abbreviations of Mark2 command names are used: `ri` for `ruleintro`, with initial `a` for `alt-` and `r` for `rev-`, `p` for `prove`, and `s` for `start`.

We develop some utilities:

```
- start "?x^+?y";

{?x ^+ ?y}

- left();

{?x} ^+ ?y

?x

- ri "?thm";

{?thm => ?x} ^+ ?y

?thm => ?x

- p "LEFT@?thm";

LEFT @ ?thm:
?x ^+ ?y =
(?thm => ?x) ^+ ?y
[]

(* the output from the following very similar proof is suppressed *)

- start "?x^+?y";
- right();
- ri "?thm";
- p "RIGHT@?thm";

RIGHT @ ?thm:
?x ^+ ?y =
?x ^+ ?thm => ?y
[]

(* output from this proof suppressed *)
- declarepretheorem "EVERYWHERE";
- start "?x.?y";
- right(); ri "EVERYWHERE@?thm"; ari "?thm";
- up();left(); ri "EVERYWHERE@?thm"; ari "?thm";
- top();
- ri "?thm";
- p "EVERYWHERE@?thm";

EVERYWHERE @ ?thm:
?x . ?y =
?thm => (?thm =>> (EVERYWHERE @ ?thm) => ?x)
. ?thm =>> (EVERYWHERE @ ?thm) => ?y
[]
```

LEFT, RIGHT, and EVERYWHERE are examples of "structural" tactics which enable the application of theorems at points other than where the tactic is actually applied. The LEFT and RIGHT tactics enable application of tactics to the immediate left and right subterms of the selecte subterm; the EVERYWHERE tactic allows a theorem to be applied to all subterms of the selected subterm in a "bottom up" fashion, if the term is built only with the . operator.

We develop an abstraction algorithm along classical lines (proof commands are given, but output is suppressed; the statement of each theorem proved is given, followed by its proof):

```
(* development of an abstraction algorithm *)

(*
ABSI @ ?x:
?x =
I . ?x
["II"]
*)

s "?x";
rri "II"; ex();
p "ABSI@?x";

(*
ABSK @ ?x:
?y =
K . ?y . ?x
["KK"]
*)

s "?y";
initializecounter();
rri "KK"; ex();
assign "?y_1" "?x";
p "ABSK@?x";

(*
ABSS @ ?x:
?T . ?x . (?U . ?x) =
S . ?T . ?U . ?x
["SS"]
*)

s "?T . ?x . (?U . ?x)";
rri "SS"; ex();
p "ABSS@?x";
```

These theorems are the basic clauses of the abstraction algorithm. The parameter that each of them takes is the term relative to which abstraction is to be carried out.

We now declare the name ABS of our abstraction tactic to prepare for its recursive definition.

```
declarepretheorem "ABS";
(* a subtactic for handling application expressions *)

(*
ABSAPP @ ?x:
?T . ?U =

(ABSS @ ?x) => ((ABS @ ?x) => ?T) . ((ABS @ ?x)
   => ?U)
["II","KK","SS"]
*)

s "?T.?U";
right(); ri "ABS@?x";
up(); left(); ri "ABS@?x";
top();
ri "ABSS@?x";
p "ABSAPP@?x";

(* the abstraction algorithm itself -- Curry's (fab) *)

(*
ABS @ ?x:
?t =

(ABSK @ ?x) =>> (ABSAPP @ ?x)
=>> (ABSI @ ?x) => ?t
["II","KK","SS"]
*)

s "?t";
ri "ABSAPP@?x";
ari "ABSI@?x";
ari "ABSK@?x";
p "ABS@?x";
```

```
(* We demonstrate why this is not a very good algorithm *)

- s "?x.(?y.?z)";
- ri "ABS@?z"; ex();
- left();
- ri "ABS@?y"; ex();
- left();
- ri "ABS@?x"; ex();
- left();
```

The display which follows is:

```
{S . (S . (K . S)
      . (S . (S . (K . S) . (S . (K . K) . (K . S)))
         . (S . (S . (K . S) . (S . (K . K) . (K
                    . K)))
            . (S . (K . K) . I))))
   . (S . (S . (K . S)
         . (S . (S . (K . S) . (S . (K . K) . (K
                    . S)))
            . (S . (S . (K . S) . (S . (K . K) . (K
                       . K)))
               . (K . I))))
      . (S . (K . K) . (K . I)))}
. ?x . ?y . ?z
```

This is not the optimal form of the composition combinator B!

```
(* the added steps needed to repair this *)

(* ABSFIX handles the idea of using K on complex terms (not just
atomic terms) which do not contain ?x; it might seem that we would
have a problem recognizing such terms, but it turns out to be easy *)

(*
ABSFIX:
S . (K . ?a) . (K . ?b) . ?x =
K . (?a . ?b) . ?x
["KK","SS"]
*)

s "S.(K.?a).(K.?b).?x";
ri "SS";
ri "EVERYWHERE@KK"; ex();
ri "ABSK@?x"; ex();
p "ABSFIX";
```

```
(* ABSFIX2 handles the idea of letting ?f rather than S(K?f)I be the
abstract from ?f.?x *)

(*
ABSFIX2:
S . (K . ?a) . I . ?x =
?a . ?x
["II","KK","SS"]
*)

s "S.(K.?a).I.?x";
ri "SS"; ri "EVERYWHERE@KK";  ri "EVERYWHERE@II"; ex();
p "ABSFIX2";

(* the following commands modify the tactic ABS to incorporate new steps *)

ae "ABS";
ri "ABSFIX";ri "ABSFIX2";
rp "ABS@?x";  (* rp is the short form of the "reprove" command *)

(*
ABS @ ?x:
?t =

ABSFIX2 => ABSFIX => (ABSK @ ?x) =>> (ABSAPP @ ?x)
=>> (ABSI @ ?x) => ?t
["II","KK","SS"]
*)

(* we repeat the development of the B combinator from above *)

- s "?x.(?y.?z)";
- ri "ABS@?z"; ex();
- left();
- ri "ABS@?y"; ex();
- left();
- ri "ABS@?x"; ex();
- left();

(* resulting display: *)

{S . (K . S) . K} . ?x . ?y . ?z

(* things come out much simpler! *)
```

We develop a reduction algorithm. The infix `*>` between theorems has the following effect: when `thm2 *> thm1` is applied, one applies `thm1`, then further applies `thm2` only in case `thm1` succeeds.

```
(* reduction algorithm *)

(*
RED:
?t =

(RED *> SS) =>> (RED *> KK) =>> (RED *> II)
=> (RIGHT @ RED) => (LEFT @ RED) => ?t
["II","KK","SS"]
*)

- declarepretheorem "RED";
- s "?t";
- ri "LEFT@RED"; ri "RIGHT@RED";
- ri "RED*>II";
- ari "RED*>KK";
- ari "RED*>SS";
- p "RED";
```

We give an example of the use of the definition facility of Mark2 (which will be further discussed below).

```
(* we define the B combinator and prove its characteristic theorem *)

defineconstant "B" "S . (K . S) . K";
```

The effect of this command is to create the following theorem:

```
B:
B =
S . (K . S) . K
["B"]

(* a theorem is developed for B analogous to the characteristic theorems
of the primitive combinators *)

(*
BB:
B . ?x . ?y . ?z =
?x . (?y . ?z)
["B","II","KK","SS"]
*)

- s "B . ?x . ?y . ?z";
- ri "EVERYWHERE@B";  (* definitional expansion followed by reduction *)
- ri "RED"; ex();
- p "BB";
```

We demonstrate the possibility of a tactic operating on lists of parameters of arbitrary length in Mark2:

```
(* we develop an even more general abstraction tool *)

- setprecedence "," 1;  (* give pairing the same left associativity as
internal application *)
- dpt "ABSLIST";
- s "?t";
- ri "ABS@?y";
- ri "LEFT@ABSLIST@?x";
- p "ABSLIST@?x,?y";

ABSLIST @ ?x , ?y:
?t =
(LEFT @ ABSLIST @ ?x) => (ABS @ ?y) => ?t
["II","KK","SS"]
```

The prover allows lists of arguments to be supplied to tactics linked by the predeclared operator `,` (ordered pair). This tactic abstracts relative to its last argument, then invokes itself with the remainder of its list of parameters as its argument. It will fail with an atomic parameter, so the first argument in the list needs to be a dummy (we always use 0). The structure of the tactic makes it a good idea to change the grouping of the `,` operator from the default.

```
(* once again, we develop the B combinator *)

- s "?x.(?y.?z)";
- ri "ABSLIST@0,?x,?y,?z"; ex();

(* this works "all at once"; now we can develop other familiar
combinators *)

(* a last example: the (suppressed) output is suggested
by the form of the following definition *)

(* the C combinator -- conversion *)

- s "?x.?z.?y";
- ri "ABSLIST@0,?x,?y,?z"; ex();
- defineconstant "C" "S . (S . (K . S) . (S . (K . K) . S)) . (K . K)";

(* its characteristic theorem *)
```

```
(*
CC:
C . ?x . ?y . ?z =
?x . ?z . ?y
["C","II","KK","SS"]
*)

- s "C.?x.?y.?z";
- ri "EVERYWHERE@C"; ri "RED"; ex(); (* definitional expansion + reduction *)
- p "CC";
```

The choice of combinatory logic as the vehicle for an extended example of the rewriting capabilities of the prover is not accidental. As we will explain in more detail below (under abstraction) the development of the tactic language was originally driven by the requirements of reduction and synthetic abstraction algorithms. Independently of this consideration, the example affords an illustration of the ability to carry out nontrivial sorts of "computation" using tactics.

The interested reader may find a more extended development of combinatory logic, including an algorithm for strong reduction, in a longer file on the author's Web page from which the example as given here is adapted.

2.8 Arithmetic and "Functional Programming"; Technical Rewriting Issues

Certain terms are rewritten by the tactic interpreter without explicit rewriting annotations. Arithmetic expressions with the predeclared operators `+!`, `*!`, `-!`, `/!`, `%!`, `<!`, and `=!` are automatically evaluated by the tactic interpreter; these are operations and relations of infinite precision unsigned integer arithmetic (the relations give boolean output (`true` or `false`)). Expressions defined by cases with an explicit boolean hypothesis are automatically collapsed: `true || ?x , ?y` is automatically rewritten to `?x` and `false || ?x , ?y` is automatically rewritten to `?y` (the predeclared operators used here are discussed below under the conditional layer).

The user may create similar effects using "functional programming" functionalities of the prover. The user may "bind" a function to a function name or an operator with instructions that this term always be applied when the function or operator is present in a term of the correct form. The form of the theorem needs to be appropriate. For example, if the user has introduced a function `car` with theorem `CAR` asserting `(car @ ?x , ?y) = ?x` and further issues the command `proveprogram "car" "CAR"`, then the tactic interpreter will automatically apply the theorem `CAR` wherever it sees the function `car` applied to a term. A preclared prefix `#` is provided which has the effect of suppressing the execution of any "functional program" attached to the top-level function or operator of its argument. There is another predeclared prefix `##` which implements laziness: rewriting annotations inside such a term are ignored unless and until a rewrite from outside the scope of the prefix affects the term.

Mark2 is quite different in its aims from other rewriting systems, to the point where it is difficult to compare them. Other rewriting systems usually involve aggressively rewriting with a list of given rules, using the rules wherever it is possible to apply them. In Mark2, the user (or a program written by the user) are supposed to indicate how rules are to be applied in detail. In standard rewriting systems, reversible rules are a problem, since obvious looping possibilities arise. In Mark2 the use of converses of rewrite rules is not a difficulty. (It is interesting to note that if the conversion infixes `<=` and `<<=` are suppressed, the logic of Mark2 changes from equational logic to precisely the "rewriting logic" proposed by the developers of OBJ (see [19]).

Mark2 can be viewed as a rewrite system of the usual sort, in which the targets of rewrite rules are not the arguments of rewriting annotations but the entire terms of form `thm => term` and other related forms (as well as arithmetic terms and those on which "functional programming" acts). One can then ask questions of familiar kinds. For example, Mark2 is "almost" confluent, if "functional programming" is ignored. The one failure of confluence is in the behaviour of the alternative infixes `=>>` and `<<=`; in a term `?thm_n =>> ... =>> ?thm_2 => ?thm_1 => term`, rewriting any subterm on the right side of a `=>>` may cause the final output to change. We regard this failure as unimportant, for a reason which can be expressed precisely: if we redefined the alternative infixes so that `(?thm_2 =>> ?thm_1) => ?term` had the behaviour now assigned to `?thm_2 =>> ?thm_1 => ?term`, then changed the precedence of the alternative infixes from 0 to 1 (which would cause them to group to the left), terms with alternative rule infixes would look and behave exactly as they do now, except that the subterms which we must avoid executing to preserve confluence now would not exist at all. We do not see any reason why a user would invoke the tactic interpreter on a term to the right of an alternative infix, so we are not inclined to make the indicated change. Another condition which needs to be noted to justify our claim that the system is essentially confluent is that the prover does not allow rewrite rules with rewriting annotations in their left sides to be applied; this is enforced by not allowing any term to match a term with rewriting annotations in it. If we view Mark2 as a conventional rewriting system, the redexes are those terms with rewriting annotations whose arguments contain no rewriting annotations; this enforces the diamond property. "Functional programming" complicates the definition of a redex somewhat.

Mark2 tactics can, of course, implement rewriting strategies of the usual more aggressive sort, as well as refinements of such strategies.

An area in which we did some work with EFTTP (the precursor of Mark2) which has not yet been upgraded to Mark2 is the investigation of "parallel execution orders" for the prover. A "breadth-first" strategy of reduction might have some advantages over the current "depth-first" approach (though speed would not be one of them, at least on a conventional machine). Such a strategy would allow the prover to carry out reductions higher up in the parse tree when it recognized that they would not interfere with reductions of terms at lower levels; this would allow some non-strict execution order (allowing some tactics to terminate which otherwise would not). An element of nondeterminism would be introduced, because theorems in lists of alternatives might be applied in preference to theorems appearing earlier in the list because it would become clear sooner that they could be applied.

2.9 Theorem Export and Theorem Search

2.9.1 Theorem Export

The reader may have noticed that every theorem proved by Mark2 is annotated with a list of names of theorems. These theorems are the axioms and definitions on which Mark2 determines that the proof of the theorem depends.

Lists of dependencies are maintained for each theorem in order to support a facility of *theorem export*: Mark2 allows theorems proved in one theory to be exported to another under suitable conditions.

The prerequisites for export of a theory from theory A to theory B is the existence of a "view" of theory B from theory A (this term is borrowed from the developers of IMPS ([7]). The view is a list of translations of names of axioms and definitions, and possibly of other symbols. It does not need to be exhaustive: Mark2 will match theorems of A with their translations into B and either reject the view and abort the export (if the theorems do not match in form) or extend the translation implied by the view as necessary. For example, if the commutativity of + in A matches the commutativity of $*$ in B, the translation table will be extended to force + to be translated by $*$; if the commutativity of + in A is reported to match a theorem in B which does not have the form of a commutative law, or if we are already committed to a different translation of +, the attempt at theorem export will fail. The export of a theorem via a view will succeed if a coherent translation of the axioms on which the theorem depends into terms of theory B can be obtained from the view. The theorem export system will export tactics just as it exports theorems; all tactics or theorems called by the tactic explicitly exported will also be exported (recursively). Name collisions are automatically avoided.

An effect of the requirements of the theorem export system is that, while the system does allow theorems to be modified in form or tactics to be debugged (by reproving a theorem/tactic of the same name) it never allows a theorem or tactic to be modified in a way which introduces additional dependencies, since this would corrupt the dependency information given about other tactics which invoke the given theorem or tactic, compromising the validity of theorem export.

The theorem export system is fairly cumbersome to invoke and has not been used extensively; it has however been valuable for exporting complex recursively defined tactics from one theory to another. We would like to improve the theorem export system to allow, for example, the user to be able to search for theorems in theories other than the one in which he/she is currently working which might be applicable to a current situation.

2.9.2 Theorem Search

The system allows the user to search for theorems in the current theory applicable to a given situation. One can ask for theorems which match a given equation. One can ask for a theorem which will convert the current term to a given form. One can ask for a list of theorems applicable to the current selected subterm. All of these features have the effect that it is not too unpleasant to work in a theory defined by another user (or by oneself a long time ago); one can discover the names of theorems one needs fairly painlessly.

Finally, most daringly, it is possible to invoke theorem searches automatically: a "theorem" of the form `?x=?y` can be executed in the tactic language, having the effect of the first theorem the prover finds in its theorem list which justifies the equation. When equations are used as theorems in a tactic, interesting effects can be achieved, since theorem searches can then be automatically set up by the tactic rather than the user, though tactics which automatically search for theorems tend to be slow. This is another feature which invites further development.

3 The Conditional Layer

This layer of the logic of Mark2 is devoted to the properties of expressions defined by cases. The canonical form for expressions of this kind is `(?a = ?b) || ?x , ?y`, which can be read "if $a = b$ then x else y". Terms of the form `?p || ?x , ?y`, where `?p` is not an equation, are understood to be equivalent to `(true = ?p) || ?x , ?y`; terms of the form `?p || ?q`, where `?q` is not a pair, are currently treated by the prover as ill-formed expressions.

In an expression of the form `?p || ?x , ?y`, we refer to `?x` and `?y` as the *cases* and to `?p` as the *hypothesis*.

The logical properties of the conditional construction which drive the handling of conditional expressions in Mark2 are the following (originally proposed as axioms for a combinatory logic implementing first-order logic with equality in the author's preprint [16]; see further discussion of this system in a later section):

projection (1): (if **true** then x else y) $= x$.

projection (2): (if **false** then x else y) $= y$.

distribution: C[if p then x else y] $=$ if p then $C[x]$ else $C[y]$, where $C[\ldots]$ represents any context.

hypothesis: (if $a = b$ then $C[a]$ else c) $=$ (if $a = b$ then $C[b]$ else c)

Auxiliary properties which handle the nature of equations and hypotheses of odd forms are:

equation: $(a = b) =$ if $a = b$ then **true** else **false**.

odd hypothesis: (if p then x else y) $=$ (if **true** $= p$ then x else y)

The role of arbitrary contexts in the distribution and hypothesis properties of the conditional was originally handled by using abstraction machinery to replace a context $C[x]$ with a function application $f(x)$. This is no longer done, for three reasons: the distribution and hypothesis properties can be adequately handled using a finite collection of instances; the abstraction system of Mark2 does not allow abstraction from every context, so an implementation using abstraction would not be fully effective or would require additional special cases; an approach based on abstraction turns out to be tactically flawed as a way to prove theorems.

The schemes of distribution and collection are replaceable by the following finite list of properties (here we use Mark2 notation, because the structure of Mark2 syntax determines the list):

positive left subterm: `(?p || (?a ^+ ?b) , ?c)` $=$ `(?p || ((?p || ?a , ?x) ^+ ?b) , ?c)`

negative left subterm: `(?p || ?a, ?b ^+ ?c)` $=$ `?p || ?a , (?p || ?x, ?b) ^+ ?c)`

positive right subterm: `(?p || (?a ^+ ?b) , ?c)` $=$ `(?p || (?a ^+ ?p || ?b, ?x) , ?c)`

negative right subterm: `(?p || ?a, ?b ^+ ?c)` $=$ `?p || ?a, ?b ^+ (?p || ?x, ?c))`

positive value: `(?p || [?a] , ?b)` $=$ `(?p || [?p || ?a , ?x] , ?b)`

negative value: `(?p || ?a, [?b])` $=$ `(?p || ?a , [?p || ?x , ?b])`

limited hypothesis: `((?a = ?b) || ?a , ?c)` $=$ `((?a = ?b) || ?b , ?c)`

case introduction: `?x` $=$ `?p || ?x , ?x`

The use of this list of properties allows one to avoid using the context $C[\ldots]$ in the hypothesis property by making use of the first six properties to bring a copy (or copies) of the hypothesis down to the level(s) of the occurrence(s) of one side of the equation which are to be replaced with the other, then applying the limited hypothesis property in place of the full hypothesis property. All instances of the distribution property can be proved by first applying case introduction to introduce a hypothesis at the top level of the term, then using the first six properties to propagate hypotheses into the term which can be used to eliminate the identical hypothesis where it occurs in subterms.

It requires a little work to show that the four properties given originally do imply the first six properties on the second list. It is sufficient to exhibit the general method of proof by proving the positive and negative value properties using the first four properties.

`?p || [?a] , ?b` $=$

(true = ?p) || [true || ?a , ?x] , ?b (by odd hypothesis and first projection) =
(true = ?p) || [?p || ?a , ?x] , ?b (by replacing true with ?p using the hypothesis property) =
?p || [?p || ?a , ?x] , ?b (by odd hypothesis) .
The same technique works for all the positive properties. The negative properties are a little trickier:
?p || ?a, [?p || ?x, ?b]
= (odd hypothesis) (true = ?p) || ?a, [?p || ?x , ?b]
= (equation) ((true = ?p) || true , false) || ?a, [?p || ?x, ?b]
= (distribution) (true = ?p) || (true || ?a , [true || ?x, ?b]) , (false || ?a , [false || ?x , ?b])
= (projection) (true = ?p) || ?a , [?b]
= (odd hypothesis) ?p || ?a, [?b].
The method of proof is similar for the other negative properties.

The equation, odd hypothesis, and case introduction hypotheses are provided as predeclared axioms by the prover. The subterm, value, and limited hypothesis properties are supported by "hard-wired" functions of the prover in a way that we will now describe.

The application of the subterm and value properties requires a strategy in which the hypothesis is duplicated ever deeper into the term being manipulated. The prover allows the application of such a strategy in effect without the syntactical cost by maintaining a list of hypotheses which are locally applicable and the senses in which they apply (positive or negative) as part of its navigation functions. The limited hypothesis property is available in the form of a "tactic" 0 |-| n, where n is a numeral, which will replace the current subterm, if it is an instance of the left side of the n-th hypothesis (which needs to be an equation), with the right side; if 0 |-| n is applied in the converse sense (using <= instead of =>), the right side of the hypothesis will be replaced with the left side. The subterm and value properties are available as a tactic 1 |-| n (with a variant 2 |-| n), which will eliminate an occurrence of the n-th hypothesis as the hypothesis of the current subterm, with the case selected to replace the subterm being determined by the sense of the n-th hypothesis. The converse will introduce a new occurrence of a hypothesis identical to the n-th hypothesis, with cases the current subterm and a new variable (or parameter supplied to the tactic, in the case of the 2 |-| n variant form), their order being determined by the sense of the n-th hypothesis.

These built-in functions are fully supported in the tactic language; they can be introduced and executed automatically by recursive tactics. In this context, a phenomenon must be noted which will be even more marked in the abstraction layer: the notion of substitution can no longer be defined in the most naive way. The refinement needed is simple in this case: a term with instances of rewriting annotations i |-| n will need to have all the indices n incremented by a suitable amount if it is substituted into a conditional expression, so as to preserve the condition that each occurrence of i |-| n is a reference to the hypothesis of the n-th conditional expression from the top of the term containing the given rewriting annotation.

An effect of the demands of the conditional layer is that the prover always rewrites the hypothesis of a case expression as far as possible before doing anything with the cases, so that it will know the correct meanings of `i |-| n`'s that it encounters in the cases. Since the tactic interpreter automatically collapses case expressions with hypothesis `true` or `false`, this means that non-strict execution can occur; a subexpression which might otherwise not terminate may be eliminated due to the reduction of a hypothesis to a boolean value. This proves useful in working with recursively defined notions.

In the discussion of the abstraction layer below, there will be found a discussion of the reasons why an attempt to implement the functions of the conditional layer in pure rewriting terms, while possible, proved very inconvenient. The sense in which we think that it fails to be a pure rewriting system is that it involves the use of built-in tactics whose meaning depends on their context (as noted above, the alternative rewriting annotation mechanism already breaks the prover as a "pure" rewriting system, but not as definitively).

3.1 Examples of Conditional Layer Functions

We first give an example of the modified notion of substitution required with the introduction of this layer:

```
- start "(?a=?b)||?x,?y";
- assign "?x" "(?c=?d)||((0|-|1)=>?c),?e";
```

Notice that the expression to be substituted for the variable `?x` contains a rewriting annotation referring to the hypothesis `?c=?d`. When we carry out this assignment, the rewriting annotation is renumbered to preserve its reference:

```
{(?a = ?b)
   || ((?c = ?d) || ((0 |-| 2) => ?c) , ?e) , ?y}
```

Normally, a rewriting annotation would not be introduced in this way (`ruleintro` or its relations would be used) but this is the easiest way to exhibit this substitution phenomenon. It usually occurs, invisibly to the user, in the course of the execution of tactics.

Since the rewriting annotation in this term refers to the equation `?c=?d` and the term to which it is applied is `?c`, its execution has an interesting effect:

```
- execute();

{(?a = ?b) || ((?c = ?d) || ?d , ?e) , ?y}
```

The first example proof is a proof that the product of two real numbers can be zero iff one of the factors in the product is zero. We begin with basic axioms relating 0, 1, multiplication, and multiplicative inverse:

```
- declareinfix "*";
- axiom "COMM2" "?x*?y" "?y*?x";
- axiom "ASSOC2" "(?x*?y)*?z" "?x*?y*?z";
- axiom "TIMESZERO" "0*?x" "0";
- declareunary "|/";
- axiom "ONE" "1*?x" "?x";
- axiom "INV" "?x* |/?x" "(0=?x)||0,1";
- axiom "ZERONOTONE" "0=1" "false";
```

The form of the axiom of multiplicative inverse deserves comment. We treat the multiplicative inverse of 0 as an otherwise unspecified real number; thus, `?x * |/?x` will have the value 0 when `?x` is 0 (the axiom `TIMESZERO` would also allow us to prove this); in other cases, we get the expected result of 1.

We display the axiom of case introduction and prove a related tactic, which allows us to supply the hypothesis as a parameter.

```
CASEINTRO:
?x =
?y || ?x , ?x
["CASEINTRO"]

(* parameterized "CASEINTRO" *)

(*
PCASEINTRO @ ?p:
?x =
?p || ?x , ?x
["CASEINTRO"]
*)

- initializecounter();
- s "?x";
- ri "CASEINTRO"; ex();
- assign "?y_1" "?p";
- prove "PCASEINTRO@?p";
```

Finally, before beginning the proof, we introduce the logical connective of disjunction by definition from the case expression primitives:

```
- defineinfix "OR" "?p|?q" "?p || true, ?q || true , false";
```

This causes the declaration of the new connective and the introduction of the following theorem:

```
OR:
?p | ?q =
?p || true , ?q || true , false
["OR"]
```

The proof follows. We will show all prover commands, but only selected prover responses. Duplicate displays of a term and an identical selected subterm will be suppressed. Comments in the proof text itself may be useful.

```
- start "0=?x*?y";
- ri "PCASEINTRO@0=?x"; ex();
```

This command introduces the hypothesis `0 = ?x`, giving the display

```
{(0 = ?x) || (0 = ?x * ?y) , 0 = ?x * ?y}
```

We now go to the case where `0 = ?x` is true:

```
- right();left();  (* case where 0 = ?x *)
- right();left();  (* now looking at ?x *)
```

At this point we see the following:

```
(0 = ?x) || (0 = {?x} * ?y) , 0 = ?x * ?y

?x
```

Application of the built-in tactic `0 |-| 1` in its converse form will convert the selected subterm `?x` to `0`.

```
- rri "0|-|1"; ex();  (* converse changes ?x to 0 *)
- up();
- ri "TIMESZERO"; ex();  (* we now see a multiplication by 0 *)
- up();
```

We now see this:

```
(0 = ?x) || {0 = 0} , 0 = ?x * ?y

0 = 0
```

We can apply the axiom `REFLEX` seen above to convert the selected subterm to `true`, completing the proof of this case.

```
- ri "REFLEX"; ex();  (* proof of this case is complete *)
- up();right();
- ri "PCASEINTRO@0=?y"; ex(); (* the status of ?y is relevant if ?x is not 0 *)
```

At this point, we introduce the hypothesis `0 = ?y` (but only under the hypothesis that `0 = ?x` is false). We see the following:

```
(0 = ?x) || true
, {(0 = ?y) || (0 = ?x * ?y) , 0 = ?x * ?y}

(0 = ?y) || (0 = ?x * ?y) , 0 = ?x * ?y
```

We now "move" into the two cases thus defined in turn. The case where `0 = ?y` is handled in a manner virtually identical to the way in which the previous case is handled (but note the use of `0 |-| 2` instead of `0 |-| 1`).

```
- right();left();  (* the case where 0 = ?y *)
- right();right();
- rri "0|-|2"; ex();  (* we need to refer to hypothesis 2 here *)
- up();
- ri "COMM2"; ri "TIMESZERO"; ex();
- up();
- ri "REFLEX"; ex();  (* this completes the proof of this case *)
- up();right();

(* this takes us to the last case, which we need to show equal to false,
not true; where ?x and ?y are both non-zero, ?x*?y will be non-zero *)
```

We have now arrived at the last case, which is the really interesting one. We expect to be able to evaluate `0 = ?x * ?y` as `false` in this case. We do this by using the theorem `EQUATION`, which implements the "equation" auxiliary property of case expressions above, to split `0 = ?x * ?y`, then move into the case where `0 = ?x * ?y` is true, and show that under the accumulated hypotheses we can rewrite `true` into `false`; this case is contradictory. This is a standard method of proof in Mark2.

```
- ri "EQUATION"; ex();
```

We pause to give the display at this point, then resume the proof:

```
(0 = ?x) || true , (0 = ?y) || true
, {(0 = ?x * ?y) || true , false}

(0 = ?x * ?y) || true , false

(* proof resumes *)

- right();left();  (* we move to the subterm "true" *)
- initializecounter();
- rri "REFLEX"; ex();
- assign "?a_1" "?x*?y*(|/?x)* |/?y";
```

We have now rewritten the instance of `true` to which we moved to the following form:

```
(?x * ?y * (|/ ?x) * |/ ?y) = ?x * ?y * (|/ ?x)
* |/ ?y
```

We use a prover command to display the current hypotheses:

```
- lookhyps();
1 (negative):
0 = ?x
2 (negative):
0 = ?y
3 (positive):
0 = ?x * ?y
```

Our strategy is to show that we can use these hypotheses to rewrite one of the instances of ?x * ?y * (|/ ?x) * |/ ?y to 0 and the other to 1, which will enable us to rewrite the equation to false!

```
- left();
- rri "ASSOC2"; ex();
```

The selected subterm is now (?x * ?y) * (|/ ?x) * |/ ?y, after regrouping, which will allow us to move left and use hypothesis 3 followed by the axiom TIMESZERO to convert the left side of the equation to 0.

```
- left();
- rri "0|-|3"; ex();
- up();
- ri "TIMESZERO"; ex();
- up();right();   (* we move to the right side of the equation *)
- rri "ASSOC2";rri "ASSOC2"; ex();
- left();left();
- ri "COMM2"; ex();
- up();
- ri "ASSOC2"; ex();
- right();
```

The right side of the equation now has the form 0 = (?y * {?x * |/ ?x}) * |/ ?y, where the braces indicate the position of the selected subterm. The selected subterm ?x * |/ ?x is a target for the axiom INV governing the multiplicative inverse.

```
- ri "INV"; ex();
```

We pause to display the entire situation.

```
(0 = ?x) || true , (0 = ?y) || true
, (0 = ?x * ?y)
|| (0 = (?y * {(0 = ?x) || 0 , 1}) * |/ ?y)
, false

(0 = ?x) || 0 , 1
```

The built-in tactic 1 |-| 1 can now be used to collapse the selected subterm to its negative case, since its hypothesis is identical to hypothesis 1.

```
- ri "1|-|1"; ex();
- up();
- ri "COMM2"; ri "ONE"; ex();
up();
```

The right side of the equation now has the form `?y * |/ ?y`, to which we can apply the same strategy of applying `INV` and collapsing the resulting case expression (but using `1 |-| 2`, since hypothesis 2 is now the relevant one).

```
- ri "INV"; ex();
- ri "1|-|2"; ex();
- up();
```

We have now converted the equation to `0 = 1`, which an axiom allows us to rewrite to `false`.

```
- ri "ZERONOTONE"; ex();
- up();up();
```

We again display the entire situation.

```
(0 = ?x) || true , (0 = ?y) || true
, {(0 = ?x * ?y) || false , false}

(0 = ?x * ?y) || false , false
```

Rewriting with the converse of `CASEINTRO` will convert the selected subterm to `false`.

```
- rri "CASEINTRO"; ex();
- top();
```

The resulting top-level expression is `(0 = ?x) || true , (0 = ?y) || true , false`, which admits rewriting using the converse of the definition of logical disjunction given above, which gives us the final form of the theorem.

```
- rri "OR"; ex();
- prove "FACTORZERO";

FACTORZERO:
0 = ?x * ?y =
(0 = ?x) | 0 = ?y
["ASSOC2","CASEINTRO","COMM2","EQUATION","INV","ONE","OR","REFLEX",
"TIMESZERO","ZERONOTONE"]
```

This proof is rather lengthy, but it should be noted that it is on a completely "nuts-and-bolts" level; a standard proof would presume some rules of inference for propositional logic which are being handled here by explicit manipulation of case expression structures. Some of its elements could be facilitated by tactics in a richer environment; for example, tactics could carry out the algebraic regroupings in the last case in single steps.

Our final example is a proof of the commutative property of disjunction (as defined above), suggesting how case expression machinery can be used to reason in propositional logic. A complete tautology checker is an involved but not difficult tactic to develop.

```
- s "?p|?q";
- ri "OR"; ex();
- ri "PCASEINTRO@?q"; ex();
```

We introduce the starting term ?p|?q and expand the expression by introducing the hypothesis ?q. We see

```
?q || (?p || true , ?q || true , false) , ?p
   || true , ?q || true , false
```

Our strategy will be to eliminate the two occurrences of ?q as hypothesis of proper subterms of this case expression and apply obvious rewrites to the results.

```
- right();left();
- right();right();
- ri "1|-|1"; ex();
```

This is the situation after the collapse of the hypothesis ?q in the positive case. The obvious rewriting opportunity is to apply the converse of case introduction to the conditional expression with true as both of its cases.

```
?q || (?p || true , {true}) , ?p || true , ?q
|| true , false

true
```

We carry this out and proceed to the other case.

```
- up();up();
- rri "CASEINTRO"; ex();
- up();right();
- right();right();
- ri "1|-|1"; ex();
```

This is the form of the expression after the collapse of the hypothesis ?q in the negative case. This has the correct form to be the target of the converse of the definition of disjunction; we go to the top and cmplete the proof.

```
?q || true , ?p || true , {false}

false
(* the proof resumes *)
- top();
- rri "OR"; ex();
- p "ORCOMM";

ORCOMM:
?p | ?q =
?q | ?p
["CASEINTRO","OR"]
```

4 The Abstraction Layer

4.1 Definition

The simplest form of definition does not involve us in abstraction: this is a definition of a constant as seen above:

```
- defineconstant "C" "S . (S . (K . S) . (S . (K . K) . S)) . (K . K)";
```

in which a new atomic constant is introduced and given a meaning. The prover does need to check that the new definition does not involve any objects not yet declared; this enforces non-circularity, because it is also required that the object to be defined has not been declared already.

As we have seen, Mark2 does allow the definition of operators as in the example

```
- defineinfix "OR" "?p|?q" "?p || true, ?q || true , false";
```

These are actually not problematic either.

Our first encounter with the functions of the abstraction layer is when we attempt to define constants or operators with parameters, i.e., when we define functions or operators on functions, as in

```
- defineconstant "Double@?x" "?x+?x";
```

or

```
- defineinfix "ADD_FUN" "(?f ++ ?g) @ ?x" "(?f @ ?x) + ?g @ ?x"
```

The additional parameter is needed in the second example as the name of the theorem which will be generated by the definition process.

Such definitions cannot be allowed without some restrictions. Clearly we can define the negation operator of propositional logic thus:

```
- defineinfix "NOT" "~?x" "(true = ?x) || false, true";
```

(here we follow the convention of Frege that any object other than the truth value "true" should be treated as being false).

We could then (if parameterized definitions were unrestricted) make the following curious definition:

```
- defineinfix "Curry_Paradox @ ?x" "~ ?x @ ?x";
```

We would then find that `Curry_Paradox @ Curry_Paradox` was equal to its own negation by definition, from which the built-in tactics and predeclared axioms of the conditional layer would suffice to derive `true = false`.

Mark2 avoids such paradoxical definitions by a restriction on abstraction in general which is perhaps easiest to understand in definitions (where variable binding is not involved). The restriction can be thought of as a system of types, though objects are not typed in the underlying logic of Mark2. These ideas are based ultimately on the definition of Quine's set theory "New Foundations" ([21]). We have discussed their application to a λ-calculus in our [15].

The type system to which we make reference is one in which the types are indexed by the natural numbers. The intended relationship between successible types is that type $n + 1$ is the type of functions with type n input and type n output. Every Mark2 operator is assigned type information in the form of a pair of integers called its "left type" and "right type". The meaning of these numbers is that if an operator `+` has left type L and right type R, and a term `T + U` is assigned type N, then `T` must be assigned type $N + L$ and `U` must be assigned type $N + R$. The intended relation of the type scheme to function application dictates that the left type of `@` be 1 and its right type 0. Similarly, the relation of the type scheme to function application indicates that if `[T]` is assigned type N, `T` should be assigned type $N - 1$. The ordered pair and equality operators are assigned left and right types of 0, as is the conditional-building infix.

A term is said to be *stratified* if and only if each free variable in the term is assigned the same type wherever it appears (when the entire equation is assigned any fixed type). A definition type-checks correctly in Mark2 if the equation term representing the theorem proposed as a definition is stratified. For example, the theorem `ADD_FUN` proposed as a definition above is legitimate: if one assigns type 0 to the whole equation `((?f ++ ?g) @ ?x) = (?f @ ?x) + ?g @ ?x` then one assigns type 0 to `?x` and type 1 to `?f` and `?g` wherever they appear. One assumes here that `+` has left and right type 0; one knows that `++` will have left and right type 0 because it is introduced by the `declareinfix` command. Special `declaretypeinfix` and `definetypedinfix` commands are needed to declare or define operators with nonzero left and/or right types. The definition of `Curry_Paradox` is not stratified: there is no way to assign a single type to `?x` in the expression `?x @ ?x`, which is enough to show that there is no way to do this in the full proposed defining theorem.

Although a notion of "relative type" of terms is employed in definitions, there are no type restrictions on the formation of Mark2 terms. `?x @ ?x` is a perfectly valid term in Mark2, though it cannot be defined as a function of `?x`.

We will see in later sections that some refinements of the notion of stratification as described here will be needed.

The freedom from paradox of permitting all function definitions which can be stratified follows from the consistency of *NFU*, the version of Quine's set theory "New Foundations" ([21]) which was shown by R. B. Jensen to be consistent relative to the usual set theory in [18]. The results of Jensen's paper are sufficient for the level of mathematical strength needed here; for the equivalence of formulations in terms of functions to formulations in terms of sets, see the author's [11], [15], [12] and [17].

4.2 Stratified λ-Calculus

It is often convenient to be able to introduce functions without having to define a new name for each function used. Mark2 supports a notation for functions using what amounts to λ-abstraction.

Up to this point, we have not allowed bound variables in our terms, and bracketed terms `[T]` have been understood to refer to constant functions with the value `T`. The referent of a bracketed term `[T]` is actually a λ-abstraction; the variable bound in a given bracket is `?1` if it is the outermost bracket in the term, `?2` if it is a proper subterm of exactly one bracket term, and, in general, `?n` if it is a proper subterm of exactly $n-1$ bracket terms. This scheme of "de Bruijn levels" (not de Bruijn indices!) is derived from de Bruijn's [5].

Furthermore, there is a stratification restriction on the formation of bracket terms: the relative type of the variable bound in a bracket term `[T]` must be the same as the type assigned to `T` everywhere that it occurs in `T` (both of these types are one less than the type assigned to `[T]` itself). The term `[~ ?1 @ ?1]` must be ill-formed for the same reason that the definition of `Curry_Paradox` above must fail.

It should be noted that the notion of stratification applied in checking the well-formedness of a bracket term is not the same as the notion of stratification applied in checking definitions; the former is a restriction on free variables, whiel the latter is a restriction on bound variables. Bracket terms appearing in definitions are required to be stratified in the sense applying to bracket terms; there is no restriction on free variables appearing with more than one type in stratified bracket terms.

The use of variable binding necessitates a further change in the definition of substitution; the use of the particular variable-binding scheme we have adopted minimizes the complexity of this change.

The required redefinition of substitution (and also of matching) is driven by the need to preserve the correlations of bound variables with their binding contexts.

Each subterm of a given term can be assigned a "level", which is the number of bracket terms of which the occurrence is a proper subterm (we are counting multiple occurrences of subterms as distinct subterms). This natural number corresponds in its role to the more complex "environment" which we would need to keep track of in an implementation of conventional variable binding. (A similar notion of level appears in the detailed implementation of substitutions involving the built-in tactics of the conditional layer).

Each subterm has a minimum level at which it can appear. For example, the term ?1 can appear only at levels ≥ 1, since the variable ?1 will otherwise have no binding context. Certain typographically identical terms have different meanings at different levels: for example, [?1] refers to the identity function at level 0, but to the constant function with value ?1 at each higher level. This can be inconvenient, and would be avoided if the scheme of deBruijn indices were used instead of deBruijn levels (or if conventional variable binding were used). The advantage of deBruijn levels over deBruijn indices is that a bound variable has the same name throughout its binding context. The advantage of either of the deBruijn schemes over conventional variable binding is the extreme simplicity of the "environment".

In a subterm appearing at level L, variables ?n with $n > L$ are bound in the subterm ("locally bound") and variables ?n with $n \leq L$ are "locally free" (they are bound in a larger context). A term appearing at a level $M > L$ will match this term iff it is the result of increasing the index of each locally bound variable by $M - L$ and leaving the indices of "locally free" variables alone. Observe that the resulting term will have no variables ?n with $L < n \leq M$. This constraint applies when we discuss the reverse direction of matching: a term appearing at a level $M < L$ will match a given term at level L iff it contains no ?n with $M < n \leq L$ and it is the result of decreasing the index of each locally bound variable by $L - M$ while leaving the indices of locally free variables alone (a locally free variable should not become locally bound in the course of a substitution). The definition of substitution follows the definition of matching: a term to be substituted into a context at level L from a context at level M is replaced in the level L context by a term that would match its occurrence at level M.

The navigation facility of Mark2 keeps track of the level of the current term, and uses this to control the matching and substitution involved in the application of rewrite rules. The implementation of the tactic language also needs to take the changing levels of terms at which rewriting annotations are generated and "executed" automatically in the course of the execution of a tactic. The `lookhyps` command of the conditional layer always displays hypotheses in the form appropriate to the current level (promoting bound variables in hypotheses so that they have the same meanings as bound variables in the currently selected subterm). Our experience is that users adapt successfully to the use of deBruijn levels in place of conventional bound variables.

Mark2 supports the semantics of bracket terms as functions with two powerful built-in tactics, `BIND` and `EVAL`. The tactic `BIND` takes a parameter: the effect of execution of `BIND @ T` is to rewrite the target term into a bracket term applied to `T`, if stratification restrictions permit. It is similar to `ABS` in the combinatory logic example above. `EVAL`, when applied to a term of the form `[T] @ U`, will carry out the indicated function application (this will always succeed). It is analogous to `RED` in the example above.

A further refinement of the definition of matching in connection with function notation has proved useful. This is a limited form of higher order matching. In a bracket in which the bound variable is ?n, a term of the form `?f @ ?n`, where

?f is a free variable, is taken to match any term T which may correspond to it in position, with ?f being taken to represent [T]. Notice that no stratification problem can arise, since a term matching ?f @ ?n in this way would necessarily be a subterm of a similar bracket term; stratification or meaningless bound variable difficulties would not arise. A simple example should clarify what is meant:

```
-  defineinfix"TIMESFUN" "(?f**?g)@?x" "(?f@?x)*?g@?x";

TIMESFUN:
(?f ** ?g) @ ?x =
(?f @ ?x) * ?g @ ?x
["TIMESFUN"]
```

The notion of multiplication of functions is defined.

```
- start "[(?f@?1)*?g@?1]@?x";
- ri "EVAL"; ex();

{EVAL => [(?f @ ?1) * ?g @ ?1] @ ?x}

{(?f @ ?x) * ?g @ ?x}

- rri "TIMESFUN"; ex();
{TIMESFUN <= (?f @ ?x) * ?g @ ?x}

{(?f ** ?g) @ ?x}

- p "TIMESFUNABSTRACT";

TIMESFUNABSTRACT:
[(?f @ ?1) * ?g @ ?1] @ ?x =
(?f ** ?g) @ ?x
["TIMESFUN"]
```

This theorem expresses an obvious relationship between products of functions and abstractions from products.

```
- s "[(?1*?1)*(?1*?1*?1)]@?x";
{[(?1 * ?1) * ?1 * ?1 * ?1] @ ?x}

- ri "TIMESFUNABSTRACT"; ex();
{TIMESFUNABSTRACT => [(?1 * ?1) * ?1 * ?1 * ?1]
   @ ?x}

{([?1 * ?1] ** [?1 * ?1 * ?1]) @ ?x}
```

The higher-order matching function of the prover is seen to operate in the fact that the prover can recognize that the theorem TIMESFUNABSTRACT applies at all: Mark2 needs to recognize the terms `?1 * ?1` and `?1 * ?1 * ?1` as being of the forms `?f @ ?1` and `?g @ ?1`, which they are not, superficially; the functions matching `?f` and `?g` are constructed by abstraction (`[?1 * ?1]` and `[?1 * ?1 * ?1]` are the functions constructed). Just as there is a change in the definition of what it is to match a term of the form `?f @ ?n`, so there is a change in the notion of substitution into a term `?f @ ?n`; if `?f` is to be replaced with an abstraction, Mark2 will replace `?f @ ?n` with the body of that abstraction (the result of removing its brackets). A simple example follows:

```
- s "[?f@?1]";

{[?f @ ?1]}

- assign "?f" "[?1*?1]";

{[?1 * ?1]}
```

This feature adds no logical strength to the prover, but it makes it possible to avoid many routine applications of `EVAL` and `BIND` which would otherwise be required.

4.3 Examples of Abstraction Layer Functions

We present some examples of the functions of this layer. First of all, we try to enter an impossible term:

```
- start "[?1@?1]";

MARK2:  Meaningless bound variable or unstratified abstraction error

{[?1 @ ?1]}
```

The prover warns that the term is impossible; it displays it but will not allow a theorem to be proved from it or even allow it to be backed up onto the desktop as a saved environment.

```
- start "?1";

MARK2:  Meaningless bound variable or unstratified abstraction error

{?1}
```

Similarly, the prover will not allow a term to be entered with a bound variable which is not bound by any bracket.

Abstraction terms are usually introduced using the `BIND` tactic:

```
- start "?x";

{?x}

- ri "BIND@?x"; ex();  (* introduce the identity function *)
{(BIND @ ?x) => ?x}

{[?1] @ ?x}

- ri "EVAL"; ex();  (* this tactic reverses the effect of BIND *)
{EVAL => [?1] @ ?x}

{?x}

- ri "BIND@?y"; ex();  (* introduce a constant function *)
{(BIND @ ?y) => ?x}

{[?x] @ ?y}
```

The term [?1] represents the identity map (when it occurs at level 0). The term [?x] represents the constant function with value ?x everywhere.

```
- start "[?x]";

{[?x]}

- assign "?x" "[?1]";

{[[?2]]}
```

This is an example of bound variable renumbering. The assignment of the value [?1] to ?x makes this object the constant function of the identity function. Clearly, the bound variable needs to be bound by the innermost set of brackets, so its index will be 2 instead of 1.

```
- start "[[?1]]";
MARK2: Meaningless bound variable or unstratified abstraction error

{[[?1]]}
```

The prover rejects the term [[?1]]. This would stand for the function which takes each object ?1 to its constant function [?1] (the K combinator of CL), but this abstraction does not type correctly in our scheme: the bound variable ?1 is assigned relative type -2 in this term, and for it to be stratified, it would have to have type -1, one lower than the type of the whole term instead of two lower.

We now attempt to construct an abstraction representing the composition operation (a relative of the B combinator).

```
- start "?f@?g@?x";

{?f @ ?g @ ?x}

- ri "BIND@?x";

{(BIND @ ?x) => ?f @ ?g @ ?x}

- ex();

{[?f @ ?g @ ?1] @ ?x}

- left();

{[?f @ ?g @ ?1]} @ ?x

[?f @ ?g @ ?1]

- ri "BIND@?g";

{(BIND @ ?g) => [?f @ ?g @ ?1]} @ ?x

(BIND @ ?g) => [?f @ ?g @ ?1]

- ex();

{[[?f @ ?1 @ ?2]] @ ?g} @ ?x

[[?f @ ?1 @ ?2]] @ ?g

- left();

({[[?f @ ?1 @ ?2]]} @ ?g) @ ?x

[[?f @ ?1 @ ?2]]
- ri "BIND@?f"; ex();

({(BIND @ ?f) => [[?f @ ?1 @ ?2]]} @ ?g) @ ?x

(BIND @ ?f) => [[?f @ ?1 @ ?2]]

({[[?f @ ?1 @ ?2]]} @ ?g) @ ?x

[[?f @ ?1 @ ?2]]
```

At this point our attempt breaks down. The relative type of `?f` in the selected subterm is 0 (the same as the type of the selected subterm itself) instead of -1 as would be required for the abstraction to succeed. The B combinator itself is not stratified. The problem can be seen in the fact that f and g are at the same type in $f(g(x))$, and at two different types in $B(f)(g)(x)$.

We take a different tack to solve the problem successfully:

```
- start "?f@?g@?x";

{?f @ ?g @ ?x}

- assign "?f" "P1@?h";

{(P1 @ ?h) @ ?g @ ?x}

- assign "?g" "P2@?h";

{(P1 @ ?h) @ (P2 @ ?h) @ ?x}
```

`P1` and `P2` are the projection operators associated with the predeclared ordered pair operator `(,)`; their definition is in the logical preamble, a collection of declarations which are automatically run when the prover is invoked.

```
- ri "BIND@?x";
{(BIND @ ?x) => (P1 @ ?h) @ (P2 @ ?h) @ ?x}

- ex();

{[(P1 @ ?h) @ (P2 @ ?h) @ ?1] @ ?x}

- left();

{[(P1 @ ?h) @ (P2 @ ?h) @ ?1]} @ ?x

[(P1 @ ?h) @ (P2 @ ?h) @ ?1]

- ri "BIND@?h"; ex();

{(BIND @ ?h) => [(P1 @ ?h) @ (P2 @ ?h) @ ?1]} @ ?x

(BIND @ ?h) => [(P1 @ ?h) @ (P2 @ ?h) @ ?1]

{[[(P1 @ ?1) @ (P2 @ ?1) @ ?2]] @ ?h} @ ?x

[[(P1 @ ?1) @ (P2 @ ?1) @ ?2]] @ ?h
```

We now define a function implementing composition based on the development above and prove that it does what we intend it to do. An `EVERYWHERE` tactic more general than the one explicitly developed above for *CL*, but with essentially the same function, is used to shorten this proof somewhat.

```
- defineconstant "Comp" "[[(P1 @ ?1) @ (P2 @ ?1) @ ?2]]";

(* defining theorem suppressed *)

- start "(Comp@?f,?g)@?x";

{(Comp @ ?f , ?g) @ ?x}

(Comp @ ?f , ?g) @ ?x

- ri "EVERYWHERE@Comp"; ex();

{(EVERYWHERE @ Comp) => (Comp @ ?f , ?g) @ ?x}

{([[(P1 @ ?1) @ (P2 @ ?1) @ ?2]] @ ?f , ?g) @ ?x}

- left(); ri "EVAL"; ex();

{[[(P1 @ ?1) @ (P2 @ ?1) @ ?2]] @ ?f , ?g} @ ?x

[[(P1 @ ?1) @ (P2 @ ?1) @ ?2]] @ ?f , ?g

{EVAL => [[(P1 @ ?1) @ (P2 @ ?1) @ ?2]] @ ?f , ?g}
@ ?x

EVAL => [[(P1 @ ?1) @ (P2 @ ?1) @ ?2]] @ ?f , ?g

{[(P1 @ ?f , ?g) @ (P2 @ ?f , ?g) @ ?1]} @ ?x

[(P1 @ ?f , ?g) @ (P2 @ ?f , ?g) @ ?1]

- up();

{[(P1 @ ?f , ?g) @ (P2 @ ?f , ?g) @ ?1] @ ?x}

- ri "EVAL"; ex();

{EVAL => [(P1 @ ?f , ?g) @ (P2 @ ?f , ?g) @ ?1]
   @ ?x}

{(P1 @ ?f , ?g) @ (P2 @ ?f , ?g) @ ?x}

- ri "EVERYWHERE@P1"; ex();
```

```
{(EVERYWHERE @ P1) => (P1 @ ?f , ?g)
   @ (P2 @ ?f , ?g) @ ?x}

{?f @ (P2 @ ?f , ?g) @ ?x}

- ri "EVERYWHERE@P2"; ex();

{(EVERYWHERE @ P2) => ?f @ (P2 @ ?f , ?g) @ ?x}

{?f @ ?g @ ?x}

- prove "Comp_Thm";

Comp_Thm:
(Comp @ ?f , ?g) @ ?x =
?f @ ?g @ ?x
["Comp","P1","P2"]
```

The construction of the `Comp` abstraction exemplifies the fact that "curried" arguments (as in $f(x)(y)$) are not interchangeable in function in the Mark2 logic with paired arguments (as in $f(x,y)$); the curried arguments x and y in the example are at different relative types, while the paired arguments are at the same relative type.

4.4 Quantification, Types and Retractions

The function of the abstraction layer is two-fold: it handles function definition, as we have already indicated, but it also handles quantification. The fluent handling of quantification necessitates a further refinement of the definition of stratification, which has the effect of providing Mark2 with logical machinery for handling data types.

The definition of the operations of first-order logic presents no difficulties. The operations of propositional logic are all definable using functions of the conditional layer alone (we have already seen definitions of disjunction and negation, from which definitions of all propositional connectives can be derived). A technical problem is that Mark2 is "applicative" in the sense that any operation can be applied to any object (this is a term introduced by Curry in [4]); thus, it is necessary to consider the effects of propositional connectives on non-truth-values, and it is sometimes necessary to use the operator `|- ?p`, defined as `?p || true , false`, to force an object to be a truth value. For example, the double negation law becomes `~ ~?p = |- ?p`.

Universal quantification is defined using abstraction as follows:

```
- defineconstant "forall@?P" "[?P@?1] = [true]";

forall:
```

```
forall @ ?P =
[?P @ ?1] = [true]
["forall"]
```

The universal quantifier is a function which, applied to a propositional function (or, in fact to any function at all) returns `true` if the target is a function whose value is `true` everywhere, and `false` if the target is a function whose value fails to be `true` somewhere (in the case of a propositional function, this would imply that its value was `false` there, but the definition needs to apply to all functions, for which this is not always the case). The existential quantifier is then easily defined.

The logical machinery already provided is adequate for reasoning in first-order logic, but the form in which the reasoning would have to be represented would be quite peculiar. The difficulty which arises is that many natural formulas of first-order logic are unstratified if represented in the most obvious way, and the circumlocutions required to avoid this problem would make the system very annoying to use.

An example makes the problem clear:

```
- start "forall@[forall@[?1=?2]]";

MARK2:  Meaningless bound variable or unstratified abstraction error
```

The term entered should be the natural way to represent $(\forall x.(\forall y.x = y))$. Equally clearly, this term is unstratified: the type assigned to `?1` would be -2, and it must be -1 for the outermost bracket to be stratified.

The reason that this does not imply a lack of strength in the logic of Mark2 is that it is possible to raise and lower types of terms known to be truth values freely. If p is a truth value, then the function application (**if** p **then** π_1 **else** π_2)(**true**,**false**) is equal to p, but the occurrence of p in this expression is at relative type 1 instead of 0. Similarly, $(\lambda x.p) = (\lambda x.\mathbf{true})$ is an expression equivalent to p (if p is a truth value) in which the relative type of p is -1. By iterating these constructions, the type of an expression known to be a truth value can be freely manipulated in order to get stratification conditions to hold. In the expression above, manipulating the type of the subexpression `forall@[?1=?2]` (raising it by one) would be sufficient to convert the whole expression to a stratified form.

However, a user of a theorem proving system should not have to carry out such operations explicitly. Experience with earlier versions of the prover convinced us that fluent reasoning with nested quantifiers would remain impossible unless the prover was given the ability to recognize expressions whose types could be raised and lowered freely, and so given the ability to recognize a wider range of terms as being stratified. In the case of truth values, the extended definition of stratification is already in place in the related set theories, in which one does not assign types to propositions at all, considering only relationships between terms within atomic propositions in determining relative types.

The solution we adopted is more general than one which simply allows truth values to be implicitly raised and lowered in type. A domain with the property that variables restricted to that domain can be freely raised or lowered in relative type is called a "strongly Cantorian set" in "New Foundations" and related systems. A set A (sets are naturally represented in Mark2 by characteristic functions) is strongly Cantorian iff the restriction of the K combinator (the operator which builds constant functions) to A is implemented by a function. In set theories, it is more natural to refer to the singleton set construction rather than the constant function construction. We discuss these ideas in detail in our [17].

Two functionalities were added to the prover, a basic functionality allowing the declaration of strongly Cantorian sets, and an auxiliary functionality making it much easier to use the basic functionality.

There is a predeclared operator `:` such that for any fixed `t`, `t : ?x` is taken to represent the result of applying a retraction to `?x` whose range is a strongly Cantorian set, the identity of which depends on `t`. The prover knows this because there is a predeclared axiom which asserts that `(t : t : ?x) = t : ?x` (applying `t` with the colon operator is a retraction) and because the definition of stratification is modified to allow the type of a term of the form `t : ?x` to be raised and lowered freely (with a uniform effect on the types of its subterms).

To solve the problem of quantification, it is sufficient to declare an atom `bool` with the defining axiom `(bool : ?p) = true = ?p`; this tells the prover that any term `bool:?p` can have its relative type freely raised and lowered. The term `forall@[bool:forall@[?1=?2]]` will be stratified.

This is the basic functionality. The term `t` in terms of the form `t : ?x` is referred to as a "type label", and that is a good indication of the way in which such terms are used. All domains commonly used as data types in computer science may be assumed to be strongly Cantorian sets, and the class (it is not a set) of strongly Cantorian sets is closed under the usual type constructors. Elsewhere (in [12]), we have argued that an identification of the notion "strongly Cantorian set" with the notion of "data type" is reasonable in the context of systems like "New Foundations".

The auxiliary functionality added to the prover which makes the basic functionality of labels for strongly Cantorian types more usable is the ability to allow the prover to recognize that certain terms belong strongly Cantorian domains even though they do not have explicit type labels. For any operator `+`, if there is a theorem of the form `?x + ?y = t : ?x + ?y`, we say that the operator is "scout" (this is short for "has strongly Cantorian output"), and a declaration can be made to the prover which will cause the prover to recognize any term of the form `?x + ?y` as capable of being freely raised or lowered in type. An operator `+` for which we have a theorem `?x + ?y = (s : ?x) + t : ?y` may be declared "scin" (short for "has strongly Cantorian input"; this stronger property allows the left and right subterms of a term with this operator to be raised or lowered in type independently of one another. Functions can also be declared "scin" or "scout" in the presence of suitable theorems: for example, the theorem `forall @ ?p = bool :  forall @ ?p` allows us to declare `forall` as "scout", and this allows the prover to recognize the problem term `forall@[forall@[?1=?2]]` as stratified without any type labels needing to be inserted. This declaration, along with declarations of the propositional connectives as "scin", completely removes all stratification problems with a natural notation for first-order logic.

This is a system of type inference with the curious feature that it is not necessary for the prover to know what type any term has; it is merely necessary for the stratification function of the prover to be able to determine that a term belongs to *some* (strongly Cantorian) type.

The enhanced definition of stratification is considerably more complex to implement, but the gain in fluency is crucial. Stratification as originally defined is a completely local property; as we will see in examples, the new definition of stratification, while it allows a broader class of terms to be recognized as stratified, makes the recognition of failures of stratification a little harder.

4.4.1 Examples of Extension of Stratification

We will work in this section with the type `bool` of booleans.

```
-declareconstant "bool";

-axiom "BOOL" "bool:?x" "true=?x";

BOOL:
bool : ?x =
true = ?x
["BOOL"]
```

These declarations introduce the boolean type.

```
start "forall@[forall@[?1=?2]]";

MARK2:  Meaningless bound variable or unstratified abstraction error
```

As above, the system will not allow us to enter this obviously legitimate quantification. We show how to remedy this situation. The crucial point is that the operator = is "scout" (its output is always of type `bool`).

```
- start "bool:?x=?y";

{bool : ?x = ?y}

- ri "BOOL"; ex();

{BOOL => bool : ?x = ?y}

{true = ?x = ?y}
```

```
- ri "EQUATION"; ex();

{EQUATION => true = ?x = ?y}

{(true = ?x = ?y) || true , false}

- rri "ODDCHOICE"; ex();

{ODDCHOICE <= (true = ?x = ?y) || true , false}

{(?x = ?y) || true , false}

- rri "EQUATION";

{EQUATION <= (?x = ?y) || true , false}

{?x = ?y}

- prove "EQBOOL";

EQBOOL:
bool : ?x = ?y =
?x = ?y
["BOOL","EQUATION","ODDCHOICE"]

- makescout "=" "EQBOOL";
```

The theorem `EQBOOL` is accepted by Mark2 as witnessing the fact that = has boolean output. The term we had trouble with above still will not work, because we also need to know that the quantifier `forall` has boolean output:

```
- s "forall@?P";

{forall @ ?P}

- ri "forall"; ex();

{forall => forall @ ?P}

{[?P @ ?1] = [true]}

- rri "EQBOOL"; ex();

{EQBOOL <= [?P @ ?1] = [true]}

{bool : [?P @ ?1] = [true]}
```

```
- right();

bool : {[?P @ ?1] = [true]}

[?P @ ?1] = [true]

- rri "forall"; ex();

bool : {forall <= [?P @ ?1] = [true]}

forall <= [?P @ ?1] = [true]

bool : {forall @ ?P}

forall @ ?P

- prove "ALLBOOL";

ALLBOOL:
forall @ ?P =
bool : forall @ ?P
["BOOL","EQUATION","ODDCHOICE","forall"]

- makescout "forall" "ALLBOOL";
```

Now we try out the offending quantification:

```
- start "forall@[forall@[?1=?2]]";

{forall @ [forall @ [?1 = ?2]]}
```

Of course, this is a false statement; one must not expect to see a proof of it! We present a proof that it is false (long, and in a dense format):

```
- ri "forall"; ex();
{forall => forall @ [forall @ [?1 = ?2]]}
{[forall @ [?1 = ?2]] = [true]}
- ri "EQUATION"; ex();
{EQUATION => [forall @ [?1 = ?2]] = [true]}
{([forall @ [?1 = ?2]] = [true]) || true , false}
- right();left();
([forall @ [?1 = ?2]] = [true]) || {true , false}
true , false
([forall @ [?1 = ?2]] = [true]) || {true} , false
true
- ri "BIND@?x"; ex();
```

```
([forall @ [?1 = ?2]] = [true])
|| {(BIND @ ?x) => true} , false
(BIND @ ?x) => true
([forall @ [?1 = ?2]] = [true]) || {[true] @ ?x}
, false
[true] @ ?x
- left();
([forall @ [?1 = ?2]] = [true]) || ({[true]} @ ?x)
, false
[true]
- rri "0|-|1"; ex();
([forall @ [?1 = ?2]] = [true])
|| ({(0 |-| 1) <= [true]} @ ?x) , false
(0 |-| 1) <= [true]
([forall @ [?1 = ?2]] = [true])
|| ({[forall @ [?1 = ?2]]} @ ?x) , false
[forall @ [?1 = ?2]]
- up();
([forall @ [?1 = ?2]] = [true])
|| {[forall @ [?1 = ?2]] @ ?x} , false
[forall @ [?1 = ?2]] @ ?x
- ri "EVAL"; ex();
([forall @ [?1 = ?2]] = [true])
|| {EVAL => [forall @ [?1 = ?2]] @ ?x} , false
EVAL => [forall @ [?1 = ?2]] @ ?x
([forall @ [?1 = ?2]] = [true])
|| {forall @ [?x = ?1]} , false
forall @ [?x = ?1]
- assign "?x" "true";
([forall @ [?1 = ?2]] = [true])
|| {forall @ [true = ?1]} , false
forall @ [true = ?1]
- ri "forall"; ex();
([forall @ [?1 = ?2]] = [true])
|| {forall => forall @ [true = ?1]} , false
forall => forall @ [true = ?1]
([forall @ [?1 = ?2]] = [true])
|| {[true = ?1] = [true]} , false
[true = ?1] = [true]
- ri "EQUATION"; ex();
([forall @ [?1 = ?2]] = [true])
|| {EQUATION => [true = ?1] = [true]} , false
EQUATION => [true = ?1] = [true]
([forall @ [?1 = ?2]] = [true])
|| {([true = ?1] = [true]) || true , false}
, false
```

```
([true = ?1] = [true]) || true , false
- right();left();
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true]) || {true , false})
, false
true , false
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true]) || {true} , false)
, false
true
- ri "BIND@false";
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true])
   || {(BIND @ false) => true} , false) , false
(BIND @ false) => true
- ex();
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true]) || {[true] @ false}
   , false)
, false
[true] @ false
- left();
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true]) || ({[true]} @ false)
   , false)
, false
[true]
- rri "0|-|2"; ex();
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true])
   || ({(0 |-| 2) <= [true]} @ false) , false)
, false
(0 |-| 2) <= [true]
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true])
   || ({[true = ?1]} @ false) , false) , false
[true = ?1]
- up();
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true])
   || {[true = ?1] @ false} , false) , false
[true = ?1] @ false
- ri "EVAL";
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true])
   || {EVAL => [true = ?1] @ false} , false)
```

```
, false
EVAL => [true = ?1] @ false
- ex();
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true]) || {true = false}
   , false)
, false
true = false
- ri "NONTRIV"; ex();
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true])
   || {NONTRIV => true = false} , false) , false
NONTRIV => true = false
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true]) || {false} , false)
, false
false
- up();
([forall @ [?1 = ?2]] = [true])
|| (([true = ?1] = [true]) || {false , false})
, false
false , false
- up();
([forall @ [?1 = ?2]] = [true])
|| {([true = ?1] = [true]) || false , false}
, false
([true = ?1] = [true]) || false , false
- rri "CASEINTRO"; ex();
([forall @ [?1 = ?2]] = [true])
|| {CASEINTRO <= ([true = ?1] = [true]) || false
   , false}
, false
CASEINTRO <= ([true = ?1] = [true]) || false
, false
([forall @ [?1 = ?2]] = [true]) || {false} , false
false
- up();up();
([forall @ [?1 = ?2]] = [true]) || {false , false}
false , false
{([forall @ [?1 = ?2]] = [true]) || false , false}
([forall @ [?1 = ?2]] = [true]) || false , false
- rri "CASEINTRO"; ex();
{CASEINTRO <= ([forall @ [?1 = ?2]] = [true])
   || false , false}
CASEINTRO <= ([forall @ [?1 = ?2]] = [true])
|| false , false
```

```
{false}
- prove "NOT_SO";
NOT_SO:
forall @ [forall @ [?1 = ?2]] =
false
["BOOL","CASEINTRO","EQUATION","NONTRIV","ODDCHOICE","forall"]
```

This is a *very* low level proof, in which all the nuts-and-bolts of all three levels are being handled "by hand". The use of the BIND and EVAL to handle instantiation of universal statements should be noted.

Finally we do a more complex stratification example. To facilitate matters, we introduce a basic property of logical conjunction (&) as an axiom.

```
- declareinfix "&";

- axiom "ANDSCIN" "?x&?y" "(bool:?x)&bool:?y";

ANDSCIN:
?x & ?y =
(bool : ?x) & bool : ?y
["ANDSCIN"]

- makescin "&" "ANDSCIN";

- definetypedinfix "IN" 0 1 "?x<<?y" "true=?y@?x";

IN:
?x << ?y =
true = ?y @ ?x
["IN"]

- s "?x<<?y";

{?x << ?y}

- ri "IN"; ex();

{IN => ?x << ?y}

{true = ?y @ ?x}

- rri "EQBOOL"; ex();

{EQBOOL <= true = ?y @ ?x}

{bool : true = ?y @ ?x}
```

```
- right();

bool : {true = ?y @ ?x}

true = ?y @ ?x

- rri "IN"; ex();

bool : {IN <= true = ?y @ ?x}

IN <= true = ?y @ ?x

bool : {?x << ?y}

?x << ?y

- p "INSCOUT";

INSCOUT:
?x << ?y =
bool : ?x << ?y
["BOOL","EQUATION","IN","ODDCHOICE"]

makescout "<<" "INSCOUT";
```

The relation << represents the membership relation (boolean-valued functions represent sets). We now develop some “set definitions” and sentences about set theory somewhat in the style of “New Foundations”.

```
- s "[?1=?1]";  (* the universal set *)

{[?1 = ?1]}

- s "[?1<<?1]";  (* complement of the Russell class *)

MARK2:  Meaningless bound variable or unstratified abstraction error

- s "[(?x<<?1)&(?1<<?x)]";  (* unobviously stratified *)

{[(?x << ?1) & ?1 << ?x]}

- s "forall@[forall@[forall@[(?1<<?2)&(?2<<?3)]]]";

(* a stratifiable assertion *)

{forall
   @ [forall @ [forall @ [(?1 << ?2) & ?2 << ?3]]]}
```

```
- s "forall@ [forall @ [forall@[(?1 << ?2) & (?2<<?3) & (?3<<?1)]]]";

(* this is unstratified (though any two of the conjuncts are OK) *)

MARK2:  Meaningless bound variable or unstratified abstraction error

{forall
   @ [forall
      @ [forall
         @ [(?1 << ?2) & (?2 << ?3) & ?3 << ?1]]]}
```

4.5 Synthetic Abstraction

An aspect of the original program of the Mark2 research which has been abandoned is an attempt to make use of synthetic abstraction instead of variable binding constructions. (We hope that we can be forgiven the minor remaining eccentricity of deBruijn level notation in place of the usual bound variables). In this section we will discuss the historical reasons why we started with a synthetic approach, the non-negligible work we did in the direction of implementing a synthetic approach, and the considerations which finally convinced us to abandon this approach. The development of the Mark2 tactic language was strongly impelled by the needs of synthetic abstraction and reduction algorithms, and the fitness of the tactic language for this purpose should be suggested by the extended example of implementation of untyped combinatory logic above.

Work on the Mark2 project was originally inspired by our definition in our Ph. D. thesis ([10]; better, see [11]) of a system of untyped synthetic combinatory logic precisely equivalent in strength and expressive power to "New Foundations". At the same time, we exhibited a version of this logic with a weakening of extensionality which has the same strength and expressive power as Jensen's *NFU* + Infinity, a mathematically adequate subsystem of "New Foundations" which is known to be consistent. The details of this system are not important in this context, because it was never directly implemented in the prover.

The system implemented in the first precursor of Mark2, the EFTTP system described in our original grant proposal [13], is a weaker system, a two-sorted synthetic combinatory logic equivalent in consistency strength and expressive power to first-order logic on infinite domains (first order logic plus sentences asserting "there are at least n objects" for each concrete natural number n). The details of this system have a great deal to do with the development of Mark2.

The system is called *EFT*, for "external function theory". It is described in our preprint [16], but most salient portions of that preprint are incorporated into this paper below. The prover which implemented it was called EFTTP; there are profound differences between EFTTP and Mark2, but an underlying family resemblance remains.

There are two sorts of term in the language of *EFT*, object terms and Function terms. Our typographical convention is that object variables and constants begin with lower-case letters, while Function variable and constants begin with upper-case letters. The intended interpretation is that the objects are the elements of some infinite set or proper class and that the Functions are the functions (possibly proper class functions) from that set into itself (or possibly a restricted class of these functions closed under suitable operations).

t and f are distinct atomic object constants (used to represent the truth values). If x and y are object terms, (x, y) is an object term, the ordered pair with projections x and y. If F is a Function term and x is an object term, $F[x]$ is an object term, the value of F at x or the result of application of F to x. Cond and Id are atomic Function terms. Id is intended to be the identity Function; $\mathrm{Cond}[(x,y),(z,w)]$ is intended to be z if $x = y$, w otherwise. If x is an object term, $|x|$ is a Function term, the constant Function of x. If F and G are Function terms, (F, G) is a Function term, the product of F and G, and $F\langle G\rangle$ is a Function term, the composition of F and G. If F is a Function term, $F!$ is a Function, called the "Hilbert Function" of F. $F![y]$ is intended to be an object such that $F[F![y], y]$ is not t, if there is any such object; its value is a matter of indifference otherwise. We write $F[x, y]$, $F\langle G, H\rangle$, instead of $F[(x, y)]$, $F\langle(G, H)\rangle$, respectively. We define the n-tuple $(x_1, x_2, ..., x_n)$ inductively as $(x_1, (x_2, ..., x_n))$.

EFT is an algebraic theory. This means that all sentences of *EFT* are equations between object terms of *EFT*. The rules of inference of *EFT* are as follows:

A. Reflexivity, symmetry, transitivity of equality.

B. If $a = c$, $b = d$ are theorems, $(a, b) = (c, d)$ is a theorem.

C. If $a = b$ is a theorem, $F[a] = F[b]$ is a theorem.

D. Uniform substitution of an object term for an object variable or of a Function term for a Function variable in a theorem yields a theorem.

Note that the rules do not directly permit substitutions of equals for equals where object terms appear as subterms of Function terms. This was reflected in the prover EFTTP by the fact that the navigation commands of the prover did not allow one to move to a Function subterm or to its object subterms.

The axioms are as follows:

(CONST) $|x|[y] = x$

(ID) $\mathrm{Id}[x] = x$

(PROD) $(F, G)[x] = (F[x], G[x])$

(COMP) $F\langle G\rangle[x] = F[G[x]]$

(PROJ1) $\mathrm{Cond}[(x, x), (y, z)] = y$

(PROJ2) $\mathrm{Cond}[(t,f),(y,z)] = z$

(DIST) $F[\mathrm{Cond}[(x,y),(z,w)] = \mathrm{Cond}[(x,y),(F[z],F[w])]$

(HYP) $\mathrm{Cond}[(x,y),(F[x],z)] = \mathrm{Cond}[(x,y),(F[y],z)]$

(HILBERT) $\mathrm{Cond}[(F[F![y],y],t),(F[x,y],t)] = t$

Axioms (PROJ1), (PROJ2), (DIST), and (HYP) should be recognizable as the basic properties of case expressions used in the discussion of the conditional layer of the prover above. Axiom (HILBERT) is used to handle quantification (by introducing an external choice operator); the other axioms are used to develop a synthetic abstraction algorithm.

It should be clear that the axioms are true in the intended interpretation (given the Axiom of Choice to support axiom (HILBERT)), and they should also serve to clarify the exact interpretations of the term constructions. If a version of the "intended interpretation" with a restricted class of functions interpreting the Functions of the theory is to be constructed, the axioms indicate the set of operations under which the restricted class of functions needs to be closed.

We have the following Abstraction Theorem:

Theorem: If s is an object term and x is an object variable which does not appear as a subterm of any Function subterm of s, there is a function term $(\lambda x)(s)$ such that x does not appear as a subterm of $(\lambda x)(s)$ and "$(\lambda x)(s)[x] = s$" is a theorem.

Proof: By induction on the structure of terms. If s is x, $(\lambda x)(s)$ is Id; if s is an atom a distinct from x, $(\lambda x)(s)$ is $|a|$. If s is of the form (u,v), $(\lambda x)(s) = ((\lambda x)(u),(\lambda x)(v))$. If s is of the form $U[v]$, U does not involve x and $(\lambda x)(s) = U\langle(\lambda x)(v)\rangle$.

$(\lambda xy)(s)$ such that "$(\lambda xy)(s)(x,y) = s$" is a theorem can be defined as $(\lambda z)(s_0)$, where z is a variable not appearing in s and s_0 is the result of replacing x with $\mathrm{Cond}[(t,t),z]$ and y with $\mathrm{Cond}[(t,f),z]$ wherever they appear in s. $(\lambda z)(\mathrm{Cond}[(t,t),z])$ and $(\lambda z)(\mathrm{Cond}[(t,f),z])$ are the projection Functions π_1 and π_2.

An interesting point about abstracts constructed following the proof of the Theorem is that they are parallel in structure to the term from which they are abstracted. This helped to make *EFT* appear to an environment in which it might be practical to avoid the use of bound variables.

Theorem: *EFT* is equivalent in deductive strength and expressive power to first-order logic with equality and the axiom scheme consisting of the assertions "there are n objects" for each concrete n (these can of course be expressed in first-order logic).

Proof: We begin the proof that *EFT* is equivalent to first-order logic with equality + "there are n objects for each concrete n" with a model-theoretic argument. Consider a first-order theory with infinite models. We construct a model of *EFT* from an infinite model of the first-order theory by letting a countably infinite model of the first order theory be the domain of objects and the set of all functions from the domain of objects into itself be the set of Functions. Since the domain of objects is infinite, there is a map from its Cartesian product with itself into itself; use any such map to define the pair of the interpretation of *EFT* (note that we could choose a map from the Cartesian product *onto* the domain of objects, getting a surjective pair). Since the model is infinite, it contains two distinct objects which can be designated as t and f. This is a case of the "intended interpretation" of *EFT* (we need the Axiom of Choice to provide functions interpreting Function terms $F!$ of *EFT*).

Observe that each predicate ϕ of the first-order theory, considered relative to a variable x, corresponds in a natural way to a function from the domain of objects (which includes the domains of interpreted n-tuples of objects for each n) to $\{t, f\}$, taking x for which ϕ holds to t and x for which $\sim \phi$ holds to f, and so can be interpreted as an *EFT* Function (we actually allow all *EFT* Functions to interpret predicates, by having values other than t or f interpret "false"). If F represents ϕ relative to x, ϕ will be represented by $F[x]$ (x may represent an n-tuple of objects as indicated below). Equality can be interpreted as $(\lambda xy)(\text{Cond}[(x,y),(t,f)])$. If p and q are objects interpreting propositions P and Q, $\text{Cond}[(p,t),(f,t)]$ interprets $\sim P$ and $\text{Cond}[((t,t),(p,q)),(t,f)]$ interprets $P\&Q$ (we recall the convention that an object other than t or f repesents the truth value "false"). Propositional logic can be interpreted in *EFT*. Thus, it is possible to interpret every quantifier-free sentence of the first-order theory. Using the abstraction theorem, we can express a quantifier-free sentence in the form $F[(x_1, x_2, x_3, ...)]$, with F containing none of the variables x_i. We now show how to quantify on x_1 and get an expression of the same form: $F[x, y] = t$ for all x exactly if $F[F![y], y] = F\langle F!, \text{Id}\rangle [y] = t$, so universal quantification with respect to x_1 yields $F\langle F!, \text{Id}\rangle [x_2, ...]$, an expression of the same form. If Neg abbreviates $(\lambda p)(\text{Cond}[(p,t),(f,t)])$, then $\text{Neg}\langle F\rangle [Neg\,\langle F\rangle![y], y] = t$ exactly if $\text{Neg}\langle F\rangle\, [x, y] = t$ for all x, so $F[\text{Neg}\langle F\rangle![y], y] = F\,\langle\text{N}eg\,\langle F\rangle!, \text{I}d\rangle\, [y] = t$ exactly if $F[x, y] = t$ for some y, and so existential quantification with respect to x_1 yields $F\,\langle\text{N}eg\,\langle F\rangle!, \text{I}d\rangle\, [x_2, ...]$, again an expression of the same form. It is thus possible to interpret all quantified sentences in prenex normal form, as long as a final dummy argument is provided in the interpretation of the quantifier-free sentence, which is easy. Every sentence in the first-order theory can thus be interpreted in the model of *EFT*.

We now consider the model theory of *EFT*. An "*EFT* theory" is defined as a set of equations in a language extending the language of *EFT* which contains the axioms and is closed under application of the rules of *EFT*. We show how to use an *EFT* theory to build a model of the intended interpretation of *EFT*, in a way which makes it clear that the interpretations of axioms and rules of first-order logic with equality are valid in *EFT*, establishing the equivalence of the two theories.

Fix an *EFT* theory T. We suppose that T has some collection of object and Function constants extending that of *EFT*. We assume that the set of terms of T can be well-ordered. The objects of the model of the intended interpretation which we will build will be equivalence classes of constant terms of T under the equivalence relation of being equated by a theorem of T. The Function terms of T will be interpreted as functions from interpreted objects to interpreted objects; to make this possible, we will interpret Function terms of the form $F!$ as actually having the form "F"!; we will assume that the exclamation-point operator acts on names of functions in T, not on functions themselves. The reason for this is that we have no guarantee that $F!$ and $G!$ will have the same extension when F and G have the same extension; the axioms ensure that the other constructions on Functions respect extension. (An alternative approach would be to add another axiom asserting that $F!$ and $G!$ have the same extension when F and G have the same extension.)

We now observe that $x =$ (by (CONST)) $|x|[\text{Cond}[(a,b),(u,v)]] =$ (by (DIST)) $\text{Cond}[(a,b),(|x|[u],|x|[v])] =$ (by (CONST)) $\text{Cond}[(a,b),(x,x)]$. We also prove the following

Lemma: If $x = y$ belongs to the minimal theory containing T and "$a = b$", then "$\text{Cond}[(a,b),(x,z)] = \text{Cond}[(a,b),(y,z)]$" belongs to T.

Proof: "$x = y$" belongs to the indicated theory if and only if there is a proof of "$x = y$" from T and "$a = b$". This proof can be presented in the form $x = w_1 = ... = w_n = y$, where w_1 is exactly x and w_n is exactly y, and each equation "$w_i = w_{i+1}$" is the result of a substitution of equals for equals based on an equation in T or the equation "$a = b$". Consider the chain of equations $\text{Cond}[(a,b),(w_i,z)] = \text{Cond}[(a,b),(w_{i+1},z)]$. If $w_i = w_{i+1}$ is based on a substitution of equals for equals on the basis of an equation in T, then this equation belongs to T. If $w_i = w_{i+1}$ is based on a substitution of a for b or vice-versa, we can use abstraction to expand it to the form $w_i = F[a] = F[b] = w_{i+1}$ (recalling that substitutions of equals for equals are only allowed by the rules of *EFT* for object terms which are not subterms of Function terms, so the abstraction theorem can be applied), and the more complex equation can be derived from a substitution instance of (HYP), so is an element of T.

This Lemma shows that all reasoning in theories extending T is actually encoded in reasoning in T itself. The converse of the Lemma is obviously true. Observe that from $t = f$ we can deduce $u = \text{Cond}[(t,t),(u,v)] = \text{Cond}[(t,f),(u,v)] = v$; a theory is universal (contains all equations) iff it contains $t = f$. We now observe that $\text{Cond}[(a,b),(u,v)] =$ (by (PROJ1) and (PROJ2))

Cond$[(a,b),($Cond$[(t,f),(v,u)]$,Cond$[(f,f),(v,u)])]$ = (apply abstraction and (DIST))
Cond$[($Cond$[(a,b),(t,f)],f),(v,u)]$.
A similar argument shows that Cond$[(a,b),(u,v)]$ = Cond$[($Cond$[(a,b),(t,f)],t),(u,v)]$.
The equation Cond$[(a,b),(t,f)] = t$ would be expected to be equivalent to $a = b$; it is easy to deduce the former from the latter; we deduce the latter from the former as follows: $b =$ (by a lemma proven above) Cond$[(a,b),(b,b)]$ = (by (HYP)) Cond$[(a,b),(a,b)]$ = (prev. paragraph) Cond$[($Cond$[(a,b),(t,f)],t),(a,b)]$ = (by hyp.) Cond$[(t,t),(a,b)] = a$. We expect the equation Cond$[(a,b),(t,f)] = f$ to correspond to the negation of $a = b$, and we see that it has at least one appropriate property; note that we can deduce Cond$[(a,b),(u,v)] = v$ from Cond$[(a,b),(t,f)] = f$ and the result above. It is clear that if $a = b$ and Cond$[(a,b),(t,f)] = f$, we can deduce $t = f$ and all other equations (by a substitution of a for b and (PROJ1)). We also observe that if an equation $c = d$ follows from T and $a = b$ and also follows from T and Cond$[(a,b),(t,f)] = f$, it follows from T alone: $c =$ Cond$[(a,b),(c,c)]$ = Cond$[(a,b),(d,c)]$ = Cond$[($Cond$[(a,b),(t,f)],f),(c,d)]$ = Cond$[($Cond$[(a,b),(t,f)],f),(d,d)]$ = d is a correct deduction in T.

We call an *EFT* theory "contradictory" iff it is the universal theory, and "consistent" if it is not contradictory (note that a theory is consistent iff it does not prove $t = f$). We can now conclude that for each consistent theory T and equation $a = b$ not in T, there is a consistent extension of T which contains either $a = b$ or Cond$[(a,b),(t,f)] = f$. From this we can deduce that each consistent theory T has a consistent extension which contains either $a = b$ or Cond$[(a,b),(t,f)] = f$ for each a and b (this requires the assumption that the set of terms can be well-ordered). We call a theory with this property "complete". We also observe that an equation is a theorem of T if and only if it holds in every complete consistent extension of T.

We now build two-valued models of the intended interpretation of T. Let T$'$ be a complete consistent extension of T. Let the objects of the model be the equivalence classes of object terms of the language of T under the relation "equated by a theorem of T$'$". Let the functions of the model be the functions from objects of the model to objects of the model induced in the obvious way by Function terms of T, recalling that the operation ! is taken to act on names of functions, not the functions themselves. Note that equality is actual equality, t and f are distinct (and as many distinct objects as desired can be built using pairing, t and f), the ordered pair is well-defined, and Id and Cond have the intended meanings, as do the product and composition operations on functions and the Hilbert operator. All such models are actually cases of the intended interpretation of *EFT*, and any equation true in all such models is a theorem of the theory T (if it were not a theorem, it would be consistent to adjoin its "negation" to T, which could be extended to a consistent complete theory). In the intended interpretation of *EFT*, the translations of theorems of first-order logic with equality given above are valid, so the translations of theorems of first-order logic with equality are valid theorems of T, and the equivalence of *EFT* and first-order logic with an external infinity of objects is established.

This proof is included here to support our claims about the logical adequacy of the conditional layer of the prover. Our implementation of quantification is different from the one used in *EFT*, since it relies on the presence of variable binding machinery.

The prover EFTTP which was the first precursor of Mark2 had term structure (and internal term data types) based exactly on the term structure of *EFT*, with an extension. A third sort of FUNCTORS was introduced: the only FUNCTOR terms are atomic constants representing operations on Functions. If FN is a FUNCTOR and F is a Function term, FN{F} is a Function term. The presence of pairing on Functions allowed the definition of FUNCTORS of more than one argument. An example of a FUNCTOR which was found useful was a functor ITER of iteration such that ITER{F,|n|} represented the Function "apply F *n* times". A further extension of the term language of EFTTP was the introduction of rewriting annotations, which were supported by a separate term construction rather than by a case of the usual operator construction as in Mark2; a result of this was that the idea of introducing parameterized tactics could not even be considered, as the data structures and parser of EFTTP did not allow for nonatomic "theorem names" (except for a unary operation for building converses of theorems).

Synthetic abstraction was very appealing in EFTTP, since the structure of Functions abstracted from terms was precisely parallel to the structure of the object term from which they were abstracted.

The reason why EFTTP was abandoned was syntactical. The syntax of *EFT* was a straitjacket on the system. The immediate motivation of the development of Mark2 was the desire to be able to use infix notation in ordinary algebra and arithmetic. In EFTTP, the associative law of addition (for example) had to be represented by

```
+[+[x?,y?],z?] = +[x?,+[y?,z?]]
```

At the same time, we observed the potentially polymorphic relationship between the sorts of object, Function, and FUNCTOR, which can be thought of as types 0, 1, and 2 in the type system underlying the relative type system of Mark2. Our already considerable familiarity with systems like "New Foundations" predisposed us to collapse these types together and adopt an untyped higher-order logic regulated by stratification rather than any absolute type scheme. Our annoyance at special term constructions went so far that we abandoned all distinctions of kind of term except that between constant functions and other terms, treating special term constructions such as ordered pair, function application, and rewriting annotation as cases of the general construction of infix terms. This added syntactical flexibility has been enormously fruitful, as for example in the development of parameterized tactics as noted above.

In primitive Mark2, the bracket construction was still employed strictly to represent constant functions, and the intention was to develop synthetic abstraction algorithms to support the (much stronger) higher order logic which had replaced the first-order logic of `EFTTP`. The only context in which stratification was actually implemented was in the definition facility; but it should be noted that the ability to define stratified functions of arbitrary order all by itself, plus the presence of an ordered pair with left and right types 0, gave the system the logical strength of *NFU* + Infinity or the theory of types of Russell, which is far stronger than first-order logic alone. (The current system is even stronger; the assumption that the set of natural numbers is strongly Cantorian, which is natural to make as soon as strongly Cantorian types are supported, gives additional strength above that of the theory of types, though still considerably weaker than the full standard set theory *ZFC*.)

Synthetic abstraction of a limited kind was developed and used in Mark2. The aim was to preserve the parallelism of structure between abstracts and terms found in *EFT*, and this can be achieved in a certain limited domain.

In the very earliest versions, the abstraction tactic needed a special subtactic for each operator over which abstraction was to be supported. This was improved by the introduction of the operation on operators represented by initial colon, which has the following defining property: for each "flat" operator (i.e., operator with both relative types zero) `+`, we have `(?f :+ ?g) @ ?x` equal to `(?f @ ?x) + (?g @ ?x)`. The colon converts a flat operator into the corresponding operation on functions. A single "built-in tactic" `RAISE` supported all instances of the property of the colon operation. The algorithm which was developed for primitive Mark2 could abstract a term relative to a variable (or other term) `?x` if the operators appearing in the term were all either function application, with no occurrence of `?x` in functions applied, or flat. The abstraction could be obtained from the term by replacing the variable `?x` with the identity function `Id`, each other atomic term outside of functions with its constant function, leaving functions alone, and replacing function application with composition; the parallelism of structure was as good as in *EFT*. However, the full abstraction capabilities of Mark2 (found at that point only in the definition facility) were considerably greater.

One indication of the unsatisfactory character of this abstraction is that it cannot be iterated. In *EFT*, considerations of sort made iteration of abstraction inconceivable, but in Mark2 it could be contemplated – but not actually carried out in an attractive way. The inability to abstract into applied functions (which makes the abstraction "predicative") was not even then reflected by any weakness of the Mark2 logic; impredicative abstractions could be introduced using the definition facility. (The prover does have a state in which abstraction of all kinds is restricted to predicative cases alone; we do not think that this is a practical restriction, though it has some interest because it implements a particular fragment of "New Foundations" which has been the object of some study).

A full synthetic abstraction facility "in principle" was obtained by extending the colon operator to apply to non-flat operators. Some care had to be taken in its definition; the relative types of `:+` for non-flat operators `+` are the same as for `+` itself, and the defining property has to be adjusted for each case of relative types to preserve stratification. We give one example: `(?f :@ ?g) @ ?x = (?f @ [?x]) @ (?g @ ?x)` (note that this permits abstraction into function applications). However, this full synthetic abstraction lost a great deal of its "readability", and lost almost all of it when it was iterated (as now became possible). Moreover, writing the full abstraction algorithm was a considerable tour-de-force, because the prover has limited ability to abstract over operators, and very limited ability to detect the relative types of operators in an automatic manner. This is a disability to be expected; operators, especially non-flat ones, are not first-class citizens of the world of the logic of Mark2.

A further reason (and really the determining reason) that synthetic abstraction was abandoned is that the practical usefulness of rewriting inside "lambda terms" became clearer and clearer as we attempted to implement even the simplest reasoning about quantifiers using synthetic abstraction. Rewriting inside λ-terms implements a weak form of extensionality which is technically difficult to implement in combinatory logic, as is already well-known to researchers in this area.

The functions of the conditional layer were not present in primitive Mark2, either, because our thinking had not yet disentangled them from an essential involvement with abstraction. The way that a conditional which would now be handled with `0 |-| 1` or its kind was handled is that a term `(?a = ?b) || ...?a... , ?c` was first converted to the form `(?a = ?b) || (?F @ ?a) , ?c` using a synthetic abstraction tactic, then converted to `(?a = ?b) || (?F @ ?b) , ?c` by the application of an equational axiom expressing (HYP), then converted to `(?a = ?b) || ...?b... , ?c` by a reduction algorithm. One of the great successes of the tactic language was that this process could be invoked as a tactic to be applied to the whole conditional expression, and it would carry out the indicated substitution without the user ever seeing an abstraction term. We did not become seriously disaffected from synthetic abstraction terms at this stage, but later on when we actually had to read them (as in implementing induction or quantification).

However, this approach to conditional rewriting has serious disadvantages. In order to apply a rewrite justified by the hypothesis of a conditional expression, it is necessary to go up to the level of the whole conditional expression. Moreover, there are serious headaches when it is desired to rewrite just one occurrence of a term which occurs in more than one place in the same case of the relevant conditional expression; this can be circumvented by marking the target to be rewritten in some syntactically distinctive way, but it is rather unpleasant to have to do it. Nonetheless, we succeeded (among other things) in writing a complete tautology checker entirely in the algebraic layer (essentially the only layer in primitive Mark2), using synthetic abstraction and reduction algorithms and axiom (HYP) to simulate the functionalities now handled by the conditional and abstraction layers of the prover.

An effect of the functions of the conditional layer which was not anticipated is that they actually strengthen the prover's logic (though inessentially). The reason for this is that, if one is limited to using abstraction in the procedure above, a hypothesis can only be applied at its own relative type. Thus, all reasoning in primitive Mark2, in the absence of unstratified user-declared axioms which enable type shifting, could be viewed as reasoning in a type theory without identifications of objects at different type levels. But the conditional layer allows substitutions justified by hypotheses to be carried out at relative types other than the type of the hypothesis; thus, unintentionally, the conditional layer was the one which actually forced the built-in logic of Mark2 to be a system like "New Foundations" instead of a merely notationally polymorphic version of type theory (this had always been the intended interpretation, and user adoption of unstratified axioms could exclude a notationally polymorphic interpretation as well). This is an inessential modification, in the sense that the logical power of a notationally polymorphic version of type theory and a genuinely untyped system with stratified abstraction are known to be the same (as long as extensionality is not assumed in the latter). But it is still an interesting observation.

4.5.1 Examples of Synthetic Abstraction

We present material from the proof script developing the predicative abstraction and reduction algorithms originally used in Mark2. Its general similarity to algorithms used in the extended combinatory logic example above should be noted.

```
(* the following  declaration ensures that the infix variable ^& will
match only flat infixes *)

declaretypedinfix 0 0 "^&";

(*
RAISE0:
(?f @ ?x) ^& ?g @ ?x =
(?f :^& ?g) @ ?x
[]
*)

- s "(?f@?x)^&(?g@?x)";
- ri "RAISE"; ex();
- p "RAISE0";

- dpt "ABSTRACT";

(*
ABSTRACT1 @ ?x:
?x =
Id @ ?x
["ID"]
*)

- s "?x";
- rri "ID";ex();
- p "ABSTRACT1@?x";

(*
ABSTRACT2 @ ?x:
?f @ ?a =
COMP <= ?f @ (ABSTRACT @ ?x) => ?a
["COMP"]
*)

- s "?f@?a";
- rri "COMP";
- right(); right(); ri "ABSTRACT@?x";
- prove "ABSTRACT2@?x";

(*
ABSTRACT3 @ ?x:
?a ^& ?b =
RAISE0 => ((ABSTRACT @ ?x) => ?a)
^& (ABSTRACT @ ?x) => ?b
[]
*)

- s "?a^&?b";
- right();
- ri "ABSTRACT@?x";
- up();left();
- ri "ABSTRACT@?x";
- top();
- ri "RAISE0";
- prove "ABSTRACT3@?x";

(*
ABSTRACT4 @ ?x:
?a =
[?a] @ ?x
[]
*)

- s "?a";
- ri "BIND@?x"; ex();
- p "ABSTRACT4@?x";

(* ABSTRACT@term will (attempt to) express a target term as a function
of its parameter "term" *)

(*
ABSTRACT @ ?x:
?a =
(ABSTRACT4 @ ?x) =>> (ABSTRACT3 @ ?x)
=>> (ABSTRACT2 @ ?x) =>> (ABSTRACT1 @ ?x) => ?a
["COMP","ID"]
*)

- s "?a";
- ri "ABSTRACT1@?x";
- ari "ABSTRACT2@?x";
- ari "ABSTRACT3@?x";
- ari "ABSTRACT4@?x";
- p "ABSTRACT@?x";

(* REDUCE will reverse the effect of ABSTRACT; it will "evaluate"
functions built by ABSTRACT *)

(*
REDUCE:
?f @ ?x =
(ABSTRACT4 @ ?x) <<= ((RL @ REDUCE) *> RAISE0)
<<= ((RIGHT @ REDUCE) *> COMP) =>> ID => ?f @ ?x
["COMP","ID"]
*)

- dpt "REDUCE";
- s "?f@?x";
- ri "ID";
- ari "(RIGHT@REDUCE)*>COMP";
- arri "(RL@REDUCE)*>RAISE0";
- arri "ABSTRACT4@?x";
- prove "REDUCE";

(* old approach to hypotheses *)
(* equational forms of tactics given without proof;
the proofs of the tactics involve no actual rewriting *)

PIVOT:
(?a = ?b) || ?T , ?U =
(RIGHT @ LEFT @ EVAL) => HYP => (?a = ?b)
```

```
|| ((BIND @ ?a) => ?T) , ?U
["HYP"]

REVPIVOT:
(?a = ?b) || ?T , ?U =
(RIGHT @ LEFT @ EVAL) => HYP <= (?a = ?b)
|| ((BIND @ ?b) => ?T) , ?U
["HYP"]
```

We now present examples of the use of these tactics.

```
- declareinfix "+"; declareinfix "*"; declareconstant "sin";

- s "2*?x+?x*?x+?y+sin@3*?x";

- ri "ABSTRACT@?x"; ex();

{([2] :* Id :+ Id :* Id :+ [?y] :+ sin @@ [3]
      :* Id)
   @ ?x}
```

It takes a little practice to see it, but the parallelism of structure here is precise. The connective `@@` represents composition of functions. Note that each atom other than `?x` is replaced with its constant function, each occurrence of `?x` with `Id`, each operator (except function application) with its type-raised version, and application with composition.

Here is an example of the old style of use of hypotheses.

```
- s "(?y=?z)||(?x+?y+?z),?w";

{(?y = ?z) || (?x + ?y + ?z) , ?w}
- ri "PIVOT"; ex();

{(?y = ?z) || (?x + ?z + ?z) , ?w}
```

Notice that the substitution has been carried out without any visible fuss.

```
- s "(?y=?z)||(?x+?y+?z),?w";

- ri "PIVOT"; steps();

{PIVOT => (?y = ?z) || (?x + ?y + ?z) , ?w}

(RIGHT @ LEFT @ EVAL) => HYP => (?y = ?z)
|| ((BIND @ ?y) => ?x + ?y + ?z) , ?w

(RIGHT @ LEFT @ EVAL) => HYP => (?y = ?z)
|| ([?x + ?1 + ?z] @ ?y) , ?w
```

```
(RIGHT @ LEFT @ EVAL) => (?y = ?z)
|| ([?x + ?1 + ?z] @ ?z) , ?w

(?y = ?z) || (LEFT @ EVAL)
=> ([?x + ?1 + ?z] @ ?z) , ?w

(?y = ?z) || (EVAL => [?x + ?1 + ?z] @ ?z) , ?w

(?y = ?z) || (?x + ?z + ?z) , ?w
```

The version of the PIVOT tactic given here actually uses the "modern" built-in abstraction and reduction instead of the synthetic algorithms. We leave to the reader's imagination what the expansion into steps would have looked like with the original abstraction and reduction.

Attempting to undo the last operation gives us a nice opportunity to illustrate the main problem with PIVOT.

```
- left();right();

{?y = ?z} || (?x + ?z + ?z) , ?w
?y = ?z

(?y = {?z}) || (?x + ?z + ?z) , ?w
?z

- rri "ID"; ex();

(?y = {ID <= ?z}) || (?x + ?z + ?z) , ?w
ID <= ?z

(?y = {Id @ ?z}) || (?x + ?z + ?z) , ?w
Id @ ?z

- top();right();left();

{(?y = Id @ ?z) || (?x + ?z + ?z) , ?w}
(?y = Id @ ?z) || {(?x + ?z + ?z) , ?w}

(?x + ?z + ?z) , ?w

(?y = Id @ ?z) || {?x + ?z + ?z} , ?w
?x + ?z + ?z

- right();left();
(?y = Id @ ?z) || (?x + {?z + ?z}) , ?w
?z + ?z
```

```
(?y = Id @ ?z) || (?x + {?z} + ?z) , ?w
?z

- rri "ID"; ex();

(?y = Id @ ?z) || (?x + {ID <= ?z} + ?z) , ?w
ID <= ?z

(?y = Id @ ?z) || (?x + {Id @ ?z} + ?z) , ?w
Id @ ?z

- top();

{(?y = Id @ ?z) || (?x + (Id @ ?z) + ?z) , ?w}

- ri "REVPIVOT"; ex();

{REVPIVOT => (?y = Id @ ?z)
   || (?x + (Id @ ?z) + ?z) , ?w}

{(?y = Id @ ?z) || (?x + ?y + ?z) , ?w}

- left();right();

{?y = Id @ ?z} || (?x + ?y + ?z) , ?w

?y = Id @ ?z

(?y = {Id @ ?z}) || (?x + ?y + ?z) , ?w

Id @ ?z

- ri "ID"; ex();

(?y = {ID => Id @ ?z}) || (?x + ?y + ?z) , ?w

ID => Id @ ?z

(?y = {?z}) || (?x + ?y + ?z) , ?w
```

In this case, where it is only desired to apply the converse of the hypothesis as a rewrite rule in one place, it is necessary to move back and forth between the top of the conditional expression and the place where the change is to be made, applying then removing syntactical markers to control the behavior of REVPIVOT. With the full functionality of the conditional layer, both of these would look precisely the same: one would apply first `0|-|1`, then its converse, in both cases at the level of the atomic term being modified. Moreover, the role of abstraction is completely eliminated.

5 Relations to Other Work

Mark2 is not closely similar to any other work in automated reasoning. We have discussed above its similarities to and differences from other work in the area of rewriting. The remoteness of Mark2 from the rewriting community is best expressed in the fact that the Knuth-Bendix algorithm and its refinements are essentially irrelevant to the Mark2 research program, though we think that it would be interesting to write an interface which would apply the Knuth-Bendix algorithm to a Mark2 theory as an automatic tactic generator. Thus, rewriting is used by Mark2 in a quite different way than it is used by a powerful automatic prover like Otter ([20])

Genetically, Mark2 is related to the group of proof systems descended from Edinburgh LCF, which have been referred to as "logical frameworks", though in a very narrow sense. We were familiar with Nuprl (though we did not have user experience) when we started work; this inspired us to write the prover in ML. But Mark2 does not manipulate proofs in the sense that Nuprl ([3]) or similar systems do (proof objects could be developed as complex constructions in the tactic language in Mark2, but this would be a very different approach from that found in the "logical frameworks" family). Mark2 is oriented toward terms rather than proofs or even propositions. Mark2 differs further from Nuprl and some other "logical frameworks" in using classical non-constructive logic. The higher-order logic based on *NFU* used by Mark2 is considerably less remote from standard mathematical practice than the complex constructive higher-order logics used by Nuprl or Coq ([6]). (This assertion may not be self-evident; it is the thesis of my paper [12] and my pending book [17]).

The prover to which Mark2 is probably most similar in overall outlook is HOL ([8]), though this is not at all obvious. HOL, though it is related to the LCF provers, has abandoned their commitment to proof objects and reasons directly in a higher order logic based on Church's simple theory of types, which is of about the same level of strength and readily mutually interpretable with the stratified higher order logic of Mark2. Both systems use classical logic. We have borrowed from HOL the idea of using the Axiom of Choice (in the form of assuming a selection operator) to facilitate reasoning with existential quantifiers. However, the superficial appearances of the systems are totally different, and HOL is primarily a manipulator of propositions, where Mark2 is primarily a manipulator of terms.

There are some incidental relationships with other systems which should be noted. We have observed above that the "rewrite logic" proposed by the developers of OBJ (in [19]) could readily be implemented in Mark2; banning the converse rewriting annotation operators (`<=`, `<<=`, and `<*`) would restrict the equational logic of Mark2 to rewriting logic. The theorem export system of

Mark2 implements the same insights as the "little theories" approach to theory modularity of the the developers of IMPS ([7]), though the implementation in IMPS is far more elegant. Though we had developed our system of theorem export already when we encountered the IMPS work, the IMPS developers' writings made it much clearer to us what we had done, and also made it clear what further developments would be necessary. Our theorem export system remains underutilized because it is still too awkward for ready use.

The features of Mark2 which appear to be really novel are its device for representing tactics as equational theorems and (oddly) its facility for easy application of rewrite rules at single locations in terms by "navigation within terms". The use of *NFU* as the higher order logic of a theorem prover is certainly novel, and the logic of case expressions used appears to be new.

6 Implementation Issues

The title of this section involves an equivocation on the meaning of "implementation" which we now explain: there is a section here on issues related to the implementation of the prover as a computer program, and a section on the implementation of mathematical theories using the prover.

6.1 Implementation of the Prover

The prover is implemented in Standard ML. The facilities of this language are naturally adapted to writing this kind of software, and we do not find anything special to say about the implementation, except that we may (accidentally, surely) have achieved the first computer implementation of Quine's definition of stratification. We say "accidentally" because stratification is a special case of the general problem of type inference, which is of course well understood; a much more complex system of type inference is found in the language ML itself, for example. Stratification is of some historical interest because "New Foundations" was one of the first polymorphic system ever defined ([21], 1937), which would make the problem of determining whether a formula is stratified one of the oldest instances of the problem of type inference.

We are interested in issues of resource use by the prover, especially issues of space. Memory management for the current version of Mark2 is entirely handled by Standard ML; in an attempt to try out some ideas for management of data structures representing terms, we have explored the idea of implementing the prover in `C++`. There is an implementation of a bare subset of the prover in `C++`, written by a student, in which theorems can be proved by hand, but which does not support the tactic language. We followed up this work (which incorporated some but not all of the memory optimizations we had in mind) with an implementation of the prover up to term display and navigation which enforces maximal sharing of memory on terms and related data types. This has not yet been upgraded to support the ability to carry out proofs, though matching functions have been written. We find ourselves very impressed with the capabilities of ML after this work. There is yet another implementation of an old version of the prover (supporting the algebraic layer and tactics), which runs on a PC and is written in Caml Light, another dialect of ML.

A project at the University of Idaho (see proposal [1]) has done some preliminary work on a graphic user interface for the prover, which seems like an obvious improvement to pursue. The ability to click a subterm of a displayed term and go there would enormously streamline the navigational functions of the prover (from the user standpoint, at least).

6.2 Implementation of Mathematical Theories

A library of elementary proof scripts is found on our Web page (address given in the introduction), covering first-order logic (both propositional and the logic of quantifiers), some elementary algebra and some elementary set theory. Preliminary investigations of induction proofs in the natural numbers are found there.

In logic, a complete tautology checker has been written as a tactic. There is a set of tools which allows the emulation of tableau proofs, but the most promising approach to doing proofs in logic (on all levels) seems to be the implementation of the equational approach of Gries exemplified in [9] and applied to program verification by Cohen in [2]. There is a file with basic declarations for an implementation of Cohen's approach to program verification on the Web page.

Our strongest evidence for the serviceability of the prover for projected applications in software verification and development from specifications has been the success of students in learning to use it and using it to develop bodies of theory. A undergraduate computer science student at Boise State, Michael Parvin, has developed the results in propositional logic and quantification found in [9]. Parvin's propositional logic work in particular has been found serviceable in the development of further theories. A group of graduate students at the University of Idaho is continuing this work under another grant from the Army Research Office (see the proposal [1]).

We have also posted a file with technical developments in untyped combinatory logic; we are testing the applicability of Mark2 as a research tool for investigations of such concepts as strong reduction in this area of logic.

7 Progress Made Under This Grant

The benefits to this work of the financial assistance of the Army Research Office have been inestimable (and the continuing support of the ARO for theory development by students at the University of Idaho as proposed in [1] is appreciated). The grant proposal was written from the standpoint of the EFTTP system. By the time funding began, the transition to the primitive Mark2 system had already been made. We were still committed to a program of using synthetic abstraction in place of variable binding constructions.

The construction of mathematical theories formally verified by machine is an extremely time-consuming task. This is especially true when the platform itself is undergoing development; actual theory development needs to be attempted to see what changes in the platform are needed. In this project, serious problems with reasoning with quantifiers were revealed by our essays in theory development: eliminating these was a multi-step process, requiring first the introduction of explicit variable binding, then the limited higher-order matching, and finally the development of facilities for automatic handling of strongly Cantorian types. This evolution was completed only fairly recently, which has retarded theory development.

Visible improvements of the prover during the period of the grant include the introduction of parameterized tactics and tactic operators, the installation of the entire conditional and abstraction layers (with the underlying prerequisite sorting out of concepts crucial to prover development), and, most recently, the implementation of strongly Cantorian types, with the "side effect" (really its primary motivation) of the fluent implementation of quantification.

We believe that the logic of the prover has reached essentially its final form, though there may be some further minor refinements. The theorem export system, and theory modularity generally, still may see major modifications (or even a fundamental shift in approach). We have some interest in developing proof objects, which would imply refinements of the tactic language, though the success of work with proof scripts has made this seem less urgent.

The C++ implementation of part of the prover is an experiment which we still may carry to its conclusion, but it is not clear whether these kinds of performance issues are crucial to the usability of the prover, since SML interpreters are now available on more platforms.

This grant has supported the work of the investigator and students in starting the construction of a body of mathematical theory adequate for applications, which continues under another ARO grant at the University of Idaho. This development work has also been the platform for user testing, which suggests that users of a wide range of levels of sophistication can learn to prove theorems with this system.

8 Appendix on Personnel

Personnel who have worked on the ARO grant have been the principal investigator, M. Randall Holmes, and a number of undergraduate research assistants. Karen Agnetta, Larry Campbell, Fongshing Lam, Michael Parvin, and Brian Mayer are or were Boise State undergraduate students who worked on the project. Of these, Michael Parvin made the greatest contribution, developing extensive proof scripts in propositional and predicate logic; Brian Mayer also made a considerable contribution (a C++ program implementing part of the prover), but was mainly funded from another source (see the next section for details).

9 Appendix on Dissemination of Results

Two papers on the Mark2 work have been published in conference proceedings so far.

These are

1. "Untyped λ-calculus with relative typing", in Dezani and Plotkin, eds., *Typed Lambda-Calculi and Applications*, the proceedings of TLCA '95, Springer, 1995.

2. "Disguising recursively chained rewrite rules as equational theorems", in Hsiang, ed., *Rewriting Techniques and Applications*, the proceedings of RTA '95, Springer, 1995.

We are planning to submit a version of this final report for publication in a suitable journal.

A monograph, *Elementary Set Theory with a Universal Set* ([17]), shortly to be published by the Cahiers series of the Center for Logic in the Department of Philosophy at the Catholic University of Louvain-la-Neuve, Belgium, acknowledges the support of this grant with respect to one chapter, which is related to theoretical issues underlying the Mark2 research. Another publication on theoretical issues which will acknowledge the support of this grant is a survey of systems of first-order logic without bound variables on which I gave a preliminary report at the 1997 Joint Mathematics Meetings in San Diego.

Two other grant proposals were funded as offshoots of this work:

1. An REU (Research Experience for Undergraduates) award through NSF EPSCoR during summer 1995, supporting a BSU computer science undergraduate who implemented an important subset of the prover in C++.

2. (with Jim Alves-Foss of the University of Idaho): "Automated Reasoning using the Mark2 Theorem Prover", Army Research Office proposal no. P-36291-MA-DPS (funded by grant no. DAAH04-96-1-0397, starting date August 1, 1996)

Development of mathematical theories under Mark2 is supported by the latter grant; this work is continuing.

References

[1] Jim Alves-Foss and M. Randall Holmes, "Automated Reasoning using the Mark2 Theorem Prover", Army Research Office proposal no. P-36291-MA-DPS (funded by grant no. DAAH04-96-1-0397, starting date August 1, 1996)

[2] Edward Cohen, *Programming in the '90's: an introduction to the calculation of programs*, Springer-Verlag, 1990.

[3] R. Constable and others, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, Englewood Cliffs, 1986.

[4] H. B. Curry and R. Feys, *Combinatory Logic*, Vol. I, North Holland, Amsterdam, 1958.

[5] N. de Bruijn, "Lambda-calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem", in Nederpelt, *et. al.*, eds., *Selected Papers on Automath*, North Holland 1994.

[6] G. Dowek *at al.*, The Coq Proof Assistant User's Guide Version 5.6. Rapport Technique 134, INRIA, December 1991.

[7] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer, "IMPS: an interactive mathematical proof system", *Journal of Automated Reasoning*, vol. 11 (1993), pp. 213-48.

[8] M. J. C. Gordon and T. F. Melham, *Introduction to HOL: a theorem proving environment for higher order logic*, Cambridge University Press, 1993.

[9] David Gries and Fred B. Schneider, *A Logical Approach to Discrete Math*, Springer-Verlag, 1993.

[10] M. Randall Holmes, "Systems of Combinatory Logic Related to Quine's 'New Foundations'", Ph.D. Dissertation, State University of New York at Binghamton, 1990.

[11] M. Randall Holmes, "Systems of Combinatory Logic Related to Quine's 'New Foundations'", *Annals of Pure and Applied Logic*, 53 (1991), pp. 103-33.

[12] Holmes, M. R. "The set-theoretical program of Quine succeeded, but nobody noticed". *Modern Logic*, vol. 4, no. 1 (1994), pp. 1-47.

[13] M. Randall Holmes, "EFTTP: an interactive equational theorem prover and programming language", Army Research Office proposal no. P-33580-MA-DPS (funded by grant no. DAAH04-93-G-0247).

[14] M. Randall Holmes, "Disguising recursively chained rewrite rules as equational theorems, as implemented in the prover EFTTP Mark 2", in *Rewriting Techniques and Applications* (proceedings of RTA '95), Springer, 1995, pp. 432-7.

[15] M. Randall Holmes, "Untyped λ-calculus with relative typing", in *Typed Lambda-Calculi and Applications* (proceedings of TLCA '95), Springer, 1995, pp. 235-48.

[16] M. Randall Holmes, "A Functional Formulation of First-Order Logic 'With Infinity' Without Bound Variables", preprint, available from the author.

[17] M. Randall Holmes, *Elementary Set Theory with a Universal Set*, volume 10 of the Cahiers du Centre de logique, Academia, Louvain-la-Neuve (Belgium), 241 pages, to appear, ISBN 2-87209-488-1.

[18] Ronald Bjorn Jensen, "On the consistency of a slight (?) modification of Quine's 'New Foundations'", *Synthese*, 19 (1969), pp. 250-63.

[19] Narciso Martí-Oliet and José Meseguer, "Rewriting Logic as a Logical and Semantic Framework", technical report SRI-CSL-93-05, SRI International, 1993.

[20] Larry Wos, et. al., *Automated Reasoning: introduction and applications*, 2nd ed., McGraw-Hill, 1992.

[21] W. V. O. Quine, "New Foundations for Mathematical Logic", *American Mathematical Monthly*, 44 (1937), pp. 70-80.

Mark2 Prover Documentation

M. Randall Holmes

September 4, 1997

Contents

1 Copyright Notice

2 Introduction; the Basic Session Model

Mark2 is an interactive equational theorem prover. The basic model for a session under this prover is that the user enters a term in the internal language of the prover, further restrained by declarations of constants and operations in the theory in which the user is working. This term can be thought of as the left side of an equation which the user is setting out to prove. The user will then modify this term using equational theorems already proved or given as axioms as rewrite rules until he converts the equation to the desired form. The user views a selected subterm of the current version of the right side of the equation; he may issue "movement" commands which change which subterm is selected, or he may apply theorems as rewrite rules to the selected subterm. All uses of equations as rewrite rules apply only to the selected subterm, not to its proper subterms. When the equation has assumed the form the user wants, it is "proved" as a theorem and becomes available for use as a rewrite rule.

There are some additional factors which complicate this basic model, but the broad outline has remained fixed through the development of the prover. One thing which should be clear is that this is not a model for an automated prover. It is also clear that this procedure would be very laborious without access to automation of reasoning steps.

The ability to automate reasoning steps is provided by a tactic language. A distinctive feature of the tactic language is that tactics or "programs" in this languages are represented as equations in the internal language, i.e., as theorems. The way in which this is achieved will be discussed below. This differs from the situation in provers in the tradition of Edinburgh LCF, in which theorems and tactics are objects at quite different levels written in different languages.

The logic immediately apparent to the user in this prover is simple equational or "algebraic" logic. There is a higher-order logic implicit in the definition and variable-binding procedures of the prover, which can be thought of as a polymorphic version of the simple type theory of Russell, in which no explicit type indices are ever needed. It is actually a version of Quine's New Foundations.

For precise syntax of commands mentioned in the text, look in the Command Reference section. Also look at the beginning of the Command Reference section for information on syntax restrictions imposed by ML (arguments must be quoted and unit arguments are needed for operations which don't have real arguments). The Command Reference represents an older layer of documentation than the rest of the test, so may not include descriptions of some commands referenced in the text. Later versions will be updated.

An old document with a tutorial and an even older command reference section of its own is appended; the earlier one is more reliable. Both have been updated to some extent in the course of the preparation of this document; they are organized along somewhat different lines.

3 The Internal Language

3.1 General Syntax

3.1.1 Atomic Terms

These fall into four categories:

numerals: 0 and non-zero-initial strings of digits. Subsumed under constants for most purposes in this document; the main difference is that numerals are predeclared.

constants: Strings consisting of characters from the set consisting of the letters (upper and lower case are allowed and distinct), digits, and the characters _ and ?. A numeral is not a constant, and a constant may not begin with ?. Numerals are subsumed under constants in most respects, except that they are predeclared.

free variables: As constants, except that they begin with ? and should contain in addition some non-digit.

bound variables: A nonzero numeral prefixed with a ? is a bound variable.

3.1.2 Operators

Operators may be either unary prefix or binary infix. A unary prefix operator is understood in the current version as representing the same operation as the infix with the same shape, but with a default left argument which depends in the infix. Operators fall into three classes:

constant: Strings of special characters other than ?, _ or paired forms like brackets, braces, and parentheses, additionally not beginning with ^ or :. A single ^ or : is also a constant operator.

free variable: As above, but with initial ^ and at least one other character.

type-raised: Initial : (when followed by at least one other character) represents a unary operation on operators (variable or constant) which can be iterated. Details below.

3.1.3 Compound Terms

terms constructed with operators: Terms are constructed with unary prefix or binary infix operators in the obvious way. Spaces are not needed between atomic terms and operators; the only situation where spaces are needed inside a term is when an infix is followed by a prefix (and this can be avoided by using parentheses).

The default order of operations is that all operators have the same precedence and associate to the right as far as possible. The user can set the precedence of an operator (and reset it at any time). Parentheses `( )` may be freely used in user-entered terms; unneeded parentheses are not displayed by the prover. The precedence of an operator is a non-negative integer; higher integers represent operators which bind more tightly, and the choice of left or right association for binary operators is determined by the parity of the precedence (even associates to the right, odd to the left).

As noted above, a prefix operator is always regarded as an abbreviation for a binary operator with the same shape with a default left argument depending on the operator. Default left arguments for operators can be set individually by the user.

variable binding contexts: The operation of enclosing a term in brackets `[ ]` produces a variable-binding context (a lambda-term). The variable bound in a context is determined by the depth of brackets: `?1` is bound in the outermost context, `?2` in the next to outermost, and so forth. Details of change of variables when substitution occurs are automated. There are restrictions on the "relative types" with which bound variables can occur in variable binding contexts which will be explained below; terms which do not meet these criteria will be parsed but rejected on the application of further tests.

The simplest use of the variable binding context is as a constant function constructor, when the bound variable is not present.

Limited higher-order matching is now available. At level 1 (`?1` free) the term `?f @ ?1` matches any level 1 term: "?f" will match the result of binding this term with brackets. Similarly, substitution of a function defined by a variable binding context for `?f` in this term will actually cause the "unbinding" of the function. Terms of the forms `?f@?2` have similar privileges at level 2 (`?2` free), and similarly for each level. This allows the direct proof of a theorem equivalent to the built-in theorem `VALUE`. When new variables are introduced by theorems, they are intoroduced as functions of the appropriate bound variable for the level at which they are introduced (this prevents the application of theorems which introduce new variables in cases where a stratification error would be introduced; this is an unintended side-effect, but it is fairly easy to work around by replacing theorems which introduce new variables with theorems to which the values of the new variables are supplied as parameters).

3.2 Navigation within Terms

The basic concept here is that each non-atomic term has a left and right subterm. The left and right subterms of a variable binding context are the same largest proper subterm. The left subterm of a term with a top-level prefix operator is the default left argument associated with that operator (it will be displayed in both local and global displays if it becomes the selected subterm).

The user sees the whole right term of the current version of the equation being proved, and sees the selected subterm as well. In the display of the entire term, the selected subterm is enclosed in braces { }. Braces are not currently used in any other way by the prover.

The basic movement commands are up, which changes the selected subterm to the smallest subterm having the current selected subterm as a proper subterm, `left` and `right`, which change the selected subterm to its left or right subterm, respectively, and `top`, which sets the selected subterm to the whole subterm.

More sophisticated movement commands, `upto`, `downtoleft`, and `downtoright`, move to remote subterms matching a term argument. New commands listed in the first command reference section below allow one to search for instances of left and right sides of theorems.

The simplest rewriting commands, `applythm` and `applyconvthm`, allow one to apply an equation whose name is given as an argument to the selected subterm. The choice of command determines whether the left or right side of the equation is matched to the selected subterm. These commands do not invoke the tactic interpreter; no automatic rewriting takes place. New commands `applyenv` and `applyconvenv` allow similar use of saved environments.

3.3 Declarations

Certain symbols need to be declared. If undeclared identifiers appear, the term entered will be parsed, but an error message will be displayed. Declarations occur automatically when some operations (such as proving theorems or making definitions) are executed.

constants: Atomic constants, other than numerals, need to be declared. Free variables and bound variables do not need to be declared. The `declareconstant` command is used.

constant operators: Infix operators need to be declared. An operator can be declared as unary, in which case it receives a default left argument of `defaultprefix` automatically (but can still be used as a binary infix!); an operator can also be assigned a specific default left argument, which must always be an atomic constant or numeral. An operator always has two integers associated with it, its "left type" and "right type", whose role in defining the notion of relative type will be discussed below. Infixes can be declared with given left and right types or with the default values of 0 for both. The `declareinfix` or `declaretypedinfix` command is used.

free variable operators: Free variable operators do not need to be declared unless it is desired that a left and right type be associated with them, in which case a declaration is necessary. The same commands are used as for constant operators. An operator variable with declared types will match only operators with the same types; an operator variable without declared types will match any operator. The `declareopaque` command can be used to block an untyped operator variable from being declared with a type later.

There are various kinds of special declaration which will be discussed elsewhere.

3.3.1 Relative Type and Stratification

In any term, each subterm can be assigned a *relative type* in accordance with the following procedure: the relative type of the entire term is 0; the relative type of the left/right subterm of a variable binding context is one less than the relative type of the variable-binding context; the relative type of the left (right) subterm of an operator term is obtained by adding the left (right) type of the operator to the relative type of the operator term.

A variable binding context entered by the user must have the property that the relative type of each occurrence of the variable bound in it is the same as the relative type of its left/right subterm. Terms violating this condition can be created during proofs but cannot be stored in theorems. Terms satisfying this condition for all subterms which are variable binding contexts are said to be *weakly stratified*.

A term is said to be *stratified* if it is weakly stratified and, in addition, each free variable appears with no more than one relative type. This notion is used in the definition facility.

These notions are best understood if one imagines (as is not the case) that each term has an integer type associated with it. Type n will be identified with the Cartesian product of type n and type n (relative types of the infix `,` of ordered pairing are both 0) while type $n+1$ is the type of functions from type n to type n. This explains the typing of variable binding contexts (lambda-terms) and is also illustrated by the fact that the left type of the infix `@` of function application is 1 and its right type is 0. The type theory being implemented can be thought of as a polymorphic version of simple type theory with a linear hierarchy of function types rather than set types; in fact, the types are all identified as in Quine's New Foundations. Any difficulty with the open problem of the consistency of New Foundations is avoided by not requiring that function application be extensional; New Foundations with urelements is known to be consistent.

3.3.2 Predicativity and Opacity

Certain special conditions complicate the definitions of weak stratification and stratification. Theories started with the `clearpredicative` command place the restriction on stratified or weakly stratified terms that no variables of appropriate types occur within any variable binding context or in the left (right) subterm of an operator term if the left (right) type of the operator happens to be negative. The `impredicative` command lifts this restriction for a theory. This used to be the default state of the prover, so theory files produced with old versions of the prover may be in this state.

The `declareopaque` command can be used to place additional restrictions on stratification notions. Operators may be declared "opaque"; a term built with an opaque operator should not contain any variables of the kind under consideration to be stratified or a subterm of a stratified term. The `declareopaque` command has an additional application: it can be used to block the later declaration of an untyped operator variable as typed. Operator variables which are not declared with types are treated as opaque in any case, since relative types cannot be assigned to subterms of terms built with such operators. The declareopaque command can only be used to declare a new operator; existing infixes cannot be made opaque.

In the case of weak stratification, the restrictions indicated in case of predicativity or opacity apply only to bound variables; in case of full stratification, they apply to all variables.

3.3.3 Refinements: Embedded Theorems and Type Retractions

The stratification function has recently been refined in two ways. Where embedded theorem names appear, they are checked for stratification internally but their type relative to the term to which they are attached is not checked. This is safe, since embedded theorems have no effect on term value. In addition, a predeclared infix `:` is provided which serves to generate type labels: the term `t : x` is the result of applying a retraction to `x`, determined by `t`, whose range is a "strongly Cantorian set" (a technical notion in set theories related to New Foundations). This has the effect of allowing the term to be freely raised and lowered in type. The effect of this on stratification is that one may freely raise and lower the types assigned to `x` and its subterms for purposes of stratifying the term in which it occurs; the type label should contain no type information (any variables in `t` must be restricted to strongly Cantorian sets). The only way in which types in `x` may influence stratification is that the difference between the types of two variables occurring in `x` may be fixed, and this will have to agree with the stratification of the context. This is motivated by the theoretical claim which we have made (in our paper in *Modern Logic*, vol. 4, no. 1) that "strongly Cantorian set" corresponds to "data type" in a model of computing based on this kind of set theory.

A further refinement along the same lines is that functions and infixes whose input or output is restricted to a "type" (strongly Cantorian set) (this must be witnessed by a theorem) can be declared as "scin" (for strongly Cantorian input) or "scout" (for strongly Cantorian output). Stratification restrictions are relaxed accordingly, without any need for explicit type labels. For example, the equality infix `=` or a universal quantifier function `forall` can be declared "scout", since their output is boolean, and so the stratification function of the prover can exploit the fact that types of boolean-valued expressions can be freely raised and lowered in order to permit more complex abstractions. This feature should have the effect of minimizing the use of explicit type labels. Note that any use of infixes declared "scin" or "scout" in abstraction terms has the effect of introducing dependencies on the witnessing theorems for those infixes.

Examples of permissible types are booleans (or any finite set) and natural numbers. Cartesian products and function spaces built from strongly Cantorian sets are strongly Cantorian. Recursive and polymorphic type construcitons will also work. The system of absolute types which can be built under Mark2 is thus as strong as that usually used in computer science.

3.3.4 The Definition Facility

The commands `defineconstant`, `defineinfix`, `definetypedinfix`, and `defineopaque` can be used to define (and automatically declare) constants or operators.

The predeclared operators `@` (function application) and `,` (ordered pair) have a special role in the definition facility.

A definition introduces an equation whose left side must have a special form.

We first define the class of *argument lists*. A free variable or iterated constant function of a free variable (built using brackets) is an argument list. Any pair (built using the `,` infix) of argument lists is an argument list. All argument lists are constructed in this way.

If a constant is being defined, the constant itself is a possible left side for its defining equation. If any term is an admissable left side, the term built by applying that term to an argument list (using the `@` infix) is an admissable left side. All admissable left sides for constant definitions are constructed in this way.

If an operator is being defined, a unary or binary term whose left and right subterms (the left may, of course, not be present; a default left argument of `defaultprefix` is automatically declared in this case) are argument lists is an admissable left side. The result of applying an admissable left side to an argument list is an admissable left side. All admissable left sides for operator definitions are constructed in this way.

Notice that "curried" higher order functions can be implicitly defined!

The further conditions on a definition are that the constant or operator being defined be undeclared, but that all other constants appearing in the defining equation be declared, that every free variable occurring on the right side also occur on the left side (no operator variables may appear in a definition), and that the defining equation considered as a term (= is predeclared and has left and right types of 0) be stratified.

Implicit function definitions using this procedure are guaranteed to avoid paradox (they are valid in the context of New Foundations with urelements). The ability to apply the constant function operator to variables allows more control over the relative typing of arguments; attention to relative type is also important when currying is used (a curried function is *not* equivalent to a similar-looking function with an argument list built using pairing; relative types are quite different!).

The `defineitconstant`, `defineitinfix` and `defineittypedinfix` commands allow one to use the selected subterm of the right side of the current equation as the right side of a definition.

4 The Tactic Language

The tactic language of Mark2 is actually a predeclared part of the internal language. Tactics are represented as equations in theories, and this is achieved via a device for representing theorems and operations on theorems as objects in the internal language of each theory.

4.1 Predeclared Operators

The predeclared operators which implement the tactic language are the *rule infixes* `=>`, `<=`, `=>>` and `<<=`. There are additional related operators `*>`, `<*` and `!!` which will be explained below. The operators `@` and `,` of function application and pairing have special roles in the tactic language as well. The predeclared operator `=` can be used as a theorem constructor. There are special unary "delay" operators `#` and `##` explained below.

There are also predeclared mathematical operators `+!`, `*!`, `-!`, `/!`, `%!`, `<!`, and `=!` of unlimited precision unsigned integer arithmetic which are executed by the tactic interpreter.

A new predeclared operator `:` implements type labels as retractions onto "strongly Cantorian sets".

4.2 Basic Semantics of Rule Infixes

The denotation of a term `thm => term` is the same as the denotation of the term `term`. `thm` is expected to denote a theorem which will be applied as a rewrite rule to `term` when the tactic interpreter `execute` or the tactic debugger `steps` are invoked. If it turns out that the left side of `thm` does not match the structure of `term`, its "application" will have no effect on the presentation of `term`.

The denotation of `thm <= term` is similar, except that the theorem `thm` is to be applied in the opposite sense (when `term` matches the right side of the equation represented by `thm` rather than the left side).

In both cases, the rule is applied only at the top level of `term`; it is not applied to proper subterms.

The prover commands `ruleintro`, `revruleintro` are used to introduce an application of a theorem (given as the argument of the command) to the selected subterm of the right side of the current equation. The prover command `droprule` will eliminate a theorem application at the top level (this works for all four infixes).

See above for relaxation of stratification rules where embedded theorem applications are involved.

4.3 Execution Order of the Tactic Language

When the `execute` command is issued, all embedded theorems in the selected subterm of the right side of the current equation are applied as rewrite rules. The rule is that innermost embedded theorems are applied first; this includes embedded theorems introduced by the execution of the tactic interpreter! It means that rewrite rules are always applied to terms which themselves contain no embedded theorems.

This is only a preliminary picture of the execution order; but it is needed to explain what is going on below.

The debugger `steps` carries out this process one step at a time, with some parallelism: it applies all and only innermost embedded theorems.

4.4 Semantics of Alternative Rule Infixes

The semantics of the *alternative rule infixes* `=>>` and `<<=` reflect a need for finer control of the circumstances under which rules are applied. These should appear only in contexts like `thm1 =>> thm2 =>> ...  thmn =>> firstthm => term`, where the directions of any arrows that appear can be reversed. Note that the last rule infix is *not* an alternative infix!

Once `term` becomes free of embedded theorems in the course of the execution of the tactic interpreter, an attempt is made to apply each of the theorems in the list, working from the inside out, until one applies, after which the remaining theorems linked with alternative infixes are ignored. Lists of alternative theorems linked with the standard rule infixes have the undesirable feature that more than one of the theorems might apply in an unexpected way.

The precise way in which this effect is achieved is that a theorem embedded via an alternative infix is executed only if the term to which it is applied is the result of an immediately preceding rewrite rule application at the top level which failed. Once an application succeeds, alternative rule applications are ignored (this will be iterated). This is the reason why the innermost infix must be standard; there is no preceding rule application to consider.

Theorems embedded with alternative infixes can be introduced using `altruleintro`, `altrevruleintro`. `altrule` can be used to toggle between alternative and standard infixes.

4.5 Semantics of Other Special Operators

The operators `*>` and `<*` are binary operators on theorems; the result of the application of `thm1 *> thm2`, for example, is that `thm2` is applied (in the usual direction) and then `thm1` is applied if the application of `thm2` succeeded.

The unary operator `!!` functions as a kind of "inlining" operator. The success or failure of application reported by `!!thm => term` is either failure if `thm` does not apply (this operator should normally be used with theorems which do not fail) or the success or failure of the last embedded theorem application at the top level of the term which results from the application of `thm`. The typical use of this is in case `thm` is of the form

```
?x = (thm1 => thm2 => ... thmn => ?x);
```

such a theorem is always applicable (and clearly true) and has the effect of causing a sequence of theorems to be applied; success/failure of `!!thm` would be success/failure of the application of `thm1` in this example.

The on-success commands can be used for building guarded lists of commands to be executed; the inlining operator is provided to make certain kinds of defined operations on theorems behave sensibly. They do not interact with each other too well: theorems of the form `(!! thm2) *> thm1` or `thm2 *> (!! thm1)` do not work in any useful way, but a theorem of the form `!!(thm2 *> thm1)` will work as expected. The problem is that the notion of success/failure is determined in different ways by the infixes `*>` and `<*` on the one hand and the inlining operator on the other. Using the criterion used by the guard infixes, the application of an inlined theorem *always* succeeds, though this may not be what the inlined theorem may report to a following theorem.

An equation `term1 = term2` can be applied as an embedded theorem; the tactic interpreter will search for a theorem which justifies the equation, and apply such a theorem if it finds it. This is useful if one has forgotten the name of a theorem whose form one knows; more powerful possibilities arise from this, since it allows the tactic interpreter to construct possible theorems "dynamically".

4.6 Theorems with Parameters

Theorems may have parameters; this will mean that the theorem name will be applied using `@` to one or more lists of arguments (a "list of arguments" is a substitution instance of an "argument list" as formally defined above) when it appears embedded. These arguments may represent objects of the theory or other theorems or tactics. It is possible to prove theorems whose names are operators, in which case an instance of the theorem will have possibly left and certainly right arguments and may further be applied to one or more lists of additional arguments.

Parameterized theorems allow the introduction of information which is not in the target term. They demonstrate one of the two senses in which Mark2 supports a higher-order programming language (the other is explained in the following section).

4.7 Functional Programming in the Tactic Language

It is possible to use the `proveprogram` command to bind a theorem to a constant (representing a function) or operator in the object language. The theorem must have a form similar to that which a definition of the constant or operator would have if it had one, except that variables on the left side can be specialized and new variables might be introduced. The theorem will then be automatically applied by the tactic interpreter wherever the constant or operator appears. The effect is to implement higher order functional programming in the tactic language.

The unary delay operator `#` can be used to suppress application of the theorem bound to the appropriate constant or operator in a term (if `f` were a function with a bound theorem, `#(f @ ?x)` would be a term in which "execution" of `f` (application of the associated theorem) was suppressed; if `**` were an operator to which a theorem was bound, `#(?x ** ?y)` would be a term in which its execution was suppressed. The delay operator is ignored by the matching function, and if a theorem is successfully applied to the "delayed" term, the delay operator is eliminated.

The unary operator `##` implements full laziness; the term to which it is applied does not have embedded theorems applied at all unless and until a theorem is applied to the lazy term.

Use of the delay operator proves necessary when the bound theorem has several steps or cases, and so takes the form of the introduction of a number of other theorems to be applied in turn or as alternatives to the original term.

The execution order of conditional expressions of the form `x || T , U` is that the subterm `x` has all theorem applications carried out before `T` or `U` is touched; this is a context in which "laziness" can be implemented without any use of the delay prefixes.

4.8 The Special Theorems RAISE and VALUE

The built-in theorem RAISE implements the semantics of the : unary operation on operators. In the case of "flat" operators (left and right type 0), the effect is to implement all equations like

```
(?x@?z) + (?y@?z) = (?x :+ ?y) @ ?z;
```

in other words, the effect of the colon operation is to produce a parallel operator on functions. The effect on operators with nontrivial relative type is the same in intent, but it is necessary to adjust types by appropriate introduction of constant function operators. The best way to find out how this works is by experiment. The RAISE theorem can be applied in the converse direction as well. The RAISE theorem only works on flat operators (without nontrivial left or right types) in a predicative theory.

The built-in theorem VALUE takes a parameter, and is intended to be applied to variable-binding contexts: the effect of applying VALUE @ [thm] to [term] is the same as the effect of executing [thm => term]. The reasons for the appearance of brackets around the parameter are technical. Limitations of matching into variable binding contexts originally forced this tactic to be provided as a primitive. These limitations no longer obtain; an equivalent theorem can be proved using limited higher-order matching.

4.9 The Special Theorems EVAL and BIND

These theorems implement abstraction and reduction for variable binding contexts: EVAL converts a term [?u] @ ?t by replacing the appropriate bound variable with t in u and "demoting" other bound variables as necessary. BIND @ ?t (note the parameter) applied to u converts it to the form [?U] @ ?t if the result [?U] of replacing each occurrence of t with the appropriate bound variable and "promoting" other bound variables as necessary is weakly stratified.

4.10 Arithmetic Operators

Operators of unsigned integer arithmetic listed above are applied whenever both arguments are numerals. The debugger now carries out arithmetic operations step by step; in earlier versions, it evaluated arithmetic expressions "all at once", a situation which would be easy to restore.

4.11 The Special Theorems INPUT and OUTPUT

The theorem OUTPUT causes the current subterm to be displayed. It is always regarded as succeeding. A keystroke is necessary to restart the tactic interpreter after output is read. The sense of the rule infix used with output (normal or reverse) has no effect on its function.

The theorem `INPUT` causes the list of current hypotheses to be displayed (see next section) along with the current subterm (this is all the information relevant to the effect of the input). The user is then prompted to input a theorem to be applied to the current subterm. It is applied using the rule infix used with `INPUT`

4.12 The Special Theorem ORDERED

The built-in theorem `ORDERED` has no effect on a term to which it is applied. When it is presented with an infix term, it reports success when the two top-level subterms are in order (an order is defined on terms, which is lexicographic on atoms with constants prededing variables and is determined by the rightmost element of a complex term) and failure otherwise. When presented with other kinds of terms, it reports success as a default. The theorem can be applied in conjunction with commutative laws for operations in the building of sorting algorithms.

4.13 The Hypothesis Facility

The hypothesis facility assigns a special role to expressions of the form `?x || ?y , ?z`, which are intended to denote `?y` if `?x` denotes `true` and `?z` otherwise. In other words, these are expressions defined by cases.

Expressions with infix `||` must have second argument a pair, or they will be rejected by the declaration checking functions of the prover (the user can create terms which violate this restriction, but they cannot appear in theorems). `||` now has some built-in execution behaviour: if `?x` evaluates to `true` or `false` under a tactic interpreter, the case expression will evaluate to the appropriate one of `?y` and `?z`. The subterm `?x` is handled by the tactic interpreter before the subterms `?y` or `?z`; conditional expressions can exhibit non-strict execution behavior.

If the subterm matching `?x` is actually an equation `?a = ?b`, then this equation is usable as a theorem in the context matching `?y`. Moreover, any case expression with the same equation in the position matching `?x` appearing in either the context matching `?y` or the context matching `?z` can be reduced to one of the alternatives (the left alternative inside `?y` and the right alternative inside `?z`).

A theorem of the form `0 |-| n`, where n is a positive integer, represents the nth hypothesis from the top level of the current term, in the form of an equation applicable in the context matched by `?y` in our generic case expression. A theorem of the form `1 |-| n` represents the same hypothesis in a form usable to decide case expressions with the same hypothesis. A theorem of the form `(2 |-| n)@parameter` has the same effect as `1 |-| n`, except that when applied in reverse (the only way it is likely to be applied) the parameter is used as the value in the newly introduced case. Mark2 controls what hypotheses are available in what contexts. The `lookhyp` and `lookhyps` commands can be used to look at hypotheses. Theorems with infix `|-|` appearing in theorems will have their numerals adjusted appropriately on application so that the theorems reference the intended hypotheses.

5 The Proof Environment

The first two subsections give a quick summary of proof and application of theorems which is useful in reading the previous section on the tactic langauge.

5.1 Proving Theorems

When the current equation is in the desired form, the command `prove` is issued. The argument of this command will either be a name for the theorem, in which case the theorem is simply recorded with that name, or, in the case of a parameterized theorem, it may be a "format" for the theorem, which can take exactly the same forms as the left side of a definition. In either event, the name of the theorem (which may in the latter case be an operator) cannot have been declared already and will automatically be declared as a constant or operator. Variants on the `prove` command are discussed in a later section.

Note that the definition facility provides another way to prove theorems of a special form. The `axiom` and `interaxiom` commands allow one to introduce axioms.

The `forget` command eliminates a theorem and all theorems which refer to it or depend on it. It is extremely slow, and it is not especially reliable.

5.1.1 Declaring Pretheorems

It is necessary when "proving" recursive or mutually recursive tactics to introduce embedded instances of theorems which have not yet been proved. The rule infix introduction commands do check for existence of theorems, so it is necessary to use the `declarepretheorem` command to signal the future intention of proving a theorem with the recursively introduced name.

5.2 Applying Theorems as Rewrite Rules

The appropriate side of the theorem is matched to the target subterm and the format of the theorem is matched to the format of the embedded instance; these matches have to be consistent with one another. If everything works, the target term is replaced with the appropriate instance of the other side of the theorem. New variables may be introduced by this process; such variables have automatically generated numerical suffixes to reduce the chance of unintended identifications of new variables with old ones or other new ones. The command `initializecounter` reinitializes the counter which provides the automatically generated numerals.

To enforce some degree of confluence, terms containing rule infixes or delay operators do not match anything for purposes of theorem application.

It should also be noted that the tactic interpreters are not applied to terms representing embedded theorems.

5.3 Features of the Proof Environment

The proof environment is a data structure containing the following elements:

name: The environment has a name. This may take the form of a term serving as the default format for the theorem to be proved (name and parameters). It is not necessary for the user to set the name. It is automatically set by certain commands, and it can be viewed using the `envname` command.

left side of theorem: This is often but not always the term originally entered by the user. Assignment commands affect both sides of the current equation, and it is possible to switch the two sides using the `workback` command.

right side of theorem: This is the term being viewed by the user.

position of selected subterm: A list of booleans is used to indicate the selected subterm; an integer representing the number of nested variable binding contexts in which it is found is also provided.

current dependencies: Axioms and definitions on which the proof so far depends are listed. Such information is maintained as an element of theorems and tactics as well.

The data structure representing a stored theorem has exactly the same structure, except that it does not include a "position of selected subterm", and it includes an additional component supporting the module system described below. The system takes advantage of this; saved proof environments are kept on the theorem list, though they cannot be used as theorems.

5.4 Starting Out

The basic command for starting out is the `start` command, which takes the initial version of the left side of the equation as a parameter. A variant is the `startfor` command, which has two parameters, the name or format of the theorem to be proved and the initial term.

The `getleftside` and `getrightside` commands allow one to get one side of a theorem as the initial term in a new proof session, in which the name/format environment variable will automatically be set to the theorem name or format. The `autoedit` command creates a proof session identical to that one has obtained when one proves the theorem, and sets the name/format to the name or format of the theorem.

The `getenv` command allows one to get a proof environment saved by name; `saveenv` or `backupenv` (the former takes a name/format as a parameter; the latter uses the current name/format or a default) allow one to save proof environments on the desktop.

5.5 Simple Permutations

The `workback` command interchanges the left and right sides of the equation. The `startover` and `starthere` commands start one fresh with the left or right (resp.) side of the former equation as both sides of the new equation (wiping dependencies).

5.6 Term Display

Many commands automatically display the current term and the selected subterm. One can use the `look` command to do this at any time, or the `lookhere` command to see just the selected term. The `lookback` command allows one to view the left side of the equation.

The display of terms uses a scheme of indentation meant to suggest the precedence structure of the term. This can be fine-tuned in various ways. The `setline` command will set line length. The `setdepth` command will set a limit on how deep into the parse tree subterms will be displayed; the `nodepth` command removes such a limit.

The precedence of operators can be reset at any time using the `setprecedence` command.

The current dependency list (axioms and definitions used in the proof so far) can be seen using `seedeps`.

5.7 Local Term Manipulation

The `applythm` and `applyconvthm` commands can be used to apply a theorem or its converse to the selected subterm without invoking the command interpreter. This might sometimes be useful for editing tactics, for example. The `applyenv` and `applyconvenv` commands allow similar use of saved environments.

The rule introduction commands `ruleintro`, `revruleintro`, `altruleintro` and `altrevruleintro` introduce embedded theorems (given as parameters) connected with `=>`, `<=`, `=>>`, or `<<=`, respectively. The command `droprule` removes an embedded theorem. The command `altrule` toggles an embedded theorem between standard and alternate versions. The commands `delay` and `undelay` introduce and remove the special delay operator `#`. The commands `lazy` and `unlazy` introduce and remove the full laziness infix `##`.

The special rule introduction commands `matchruleintro` and `targetruleintro` will introduce a theorem (if there is one) which matches a given equation or which converts the selected subterm to a given form, respectively.

The `execute` command invokes the tactic interpreter on the selected subterm. The `onestep` command carries out one step, executing all innermost embedded theorems in parallel. The `steps` command does one step each time the enter key is struck.

5.8 Global Variable Assignment

There is a spectrum of assignment commands which make substitutions for variables in a couple of different senses. These commands are unusual in that they can affect the left side of the current equation. The most important use of these commands is in assigning values to new variables introduced by theorem applications; another use for them is in entering large terms.

The `assign` command takes two terms as parameters and carries out the substitutions for variables represented by the match of those two terms (the simplest application is a statement like `assign ?x ?y+?z`, which would replace all occurrences of `?x` with `?y+?z`. Note that substitutions have to be carried out globally (not just in the selected term) and in both sides of the current equation!

The `assigninto` command takes as arguments a free variable term and a term, and produces a new equation which results from substituting both sides of the current equation for the free variable parameter in the term parameter; this implements "doing the same thing to both sides of an equation".

Each of these commands can be suffixed with `-it`, which causes the last parameter to be set to the selected subterm.

The `assignleft` and `assignright` commands allow one to assign the left or right side of a theorem to a variable.

5.9 Finishing Up

The basic command for completing a session is `prove`, which takes as a parameter the format of the theorem to be proved (the name plus any parameters).

If the name/format environment variable has been set (this can be ascertained using the `envname` command), the command `autoprove` will record the current equation as a theorem with the format indicated by the environment variable.

The `reprove` command allows one to modify an existing theorem, replacing it with the current equation. Normally, this is used for debugging tactics. The only constraint is that the new version of the theorem cannot have stronger dependencies than the old; if this were not enforced, the dependency information of tactics which invoke the edited theorem would be corrupted.

The `autoreprove` variant of this command is often used, because the `getleftside`, `getrightside`, and `autoedit` commands, which are often used for editing, all set the name/format environment variable.

A proof environment does not close down when a theorem is proved; the current equation remains the same until a new command in the `start` family is issued. In this case, the current equation is usually automatically backed up (the `backupenv` command is invoked).

When a new theorem environment is started up, the attempt to back up the previous theorem environment will generate an error message applying to that previous environment if there is something wrong with it.

5.10 The Module System

A theorem has an additional component not found in a proof environment: this is a list of subsidiary theorems visible only to the parent theorem. The intended use of this feature is to make it possible to avoid cluttering the theorem list with subcommands of tactics.

The `pushtheorem` command allows one to "hide" one theorem in another. It automatically comments the "pushed" theorem so that one can tell where it is (it ceases to be visible on the master theorem list). The `poptheorem` command allows one to retrieve a theorem, also posting an automatic comment. Variants `pushtheorem2` and `poptheorem2` do not post comments; this is useful if one is taking a subtactic out of a module in order to edit it, for example.

A theorem with subsidiary theorems is referred to as a "module". Modules can be nested. Theorems inside a module are only available when the parent theorem is being executed; be sure that a theorem packed into a module is not needed by any theorem other than its parent theorem or other theorems in the module! Their names are still reserved.

The `forget` command will eliminate an entire module if any component depends on the theorem being forgotten. Theorem export will work correctly in the presence of modules, but it will not export module structure; exported modules will be completely unpacked.

6 The Theory Environment

This section describes the level on which the user interacts with theorems and declarations in a fixed theory.

6.1 Components of the Theory Environment

The theory environment consists of the constant and operator declaration list, the master theorem list, and some satellite declaration lists and lists containing syntactical information. A theory has a name which associates it with a file.

Precedences: A list of operator precedences. Parity of the precedence determines left/right associativity (even gives right, odd left). Default precedence of all operators is 0.

Default left arguments: A list of default left arguments for unary operators.

Declarations: There is a list of declarations of constants, operators, and free variable operators (the last only for relative typing purposes; free variable operators which do not need to be typed do not need to be declared).

Opacity declarations: A list of operators declared opaque.

Pretheorems: A list of names of theorems which are intended to be proved; needed to make recursive tactics possible while preserving declaration checking for embedded theorems.

Definitions: A list of constants and operators with the theorems which define them; the name of the theorem will be the same as the name of the defined object, if it is a constant.

Functional programs: A list of automatic bindings of theorems to constants and operators, implementing functional programming in the tactic language.

Theorems: There is a list of theorems; this also includes images of current and old saved proof environments. The structure of an individual theorem parallels the structure of a proof environment as noted above, with the addition of the list of theorems supporting the module system. Proof environments saved on the theorem list lose "position of selected subterm" information, since this is the one item in the structure of a proof environment that is not found in a theorem.

6.2 The Syntax Lists

Use the `setprecedence` command to set precedence of an operator. Use the `declareprefix` command to set a specific default left argument for an operator; one can also declare a new unary operator with `declareunary` or declare an operator implicitly via the definition or proof commands as a unary operator, in which case it is automatically assigned a default left argument of `defaultprefix`.

The `showprecs` command will display all precedences, including the formally redundant left/right associativity information.

Declaration display commands will show default left arguments for operators which have them.

6.3 Declarations and Comments

Constants and operators can be declared using the `declareconstant`, `declareinfix` (declares flat operators), and `declaretypedinfix` (declares operators with non-trivial type) commands.

Constants and operators can also be declared implicitly via the declaration and proof commands. It should be noted that theorem names are constants of a theory! When constants are defined, the name of the defining theorem is the same as the name of the object being defined; when operators are defined, the name of the defining theorem must be given as a parameter.

A single declaration can be viewed using the `showdec` command. A declaration line contains the identifier followed by the left and right types (both 0 in the case of non-operators) and the default left argument if there is one. The `showalldecs` or `scandecs` commands can be used to view all declarations.

Declarations now also include comments. New comments (superseding any previous comments) can be attached to a declared object using the `newcomment` command; the `appendcomment` command appends comments to any previously existing comments. `showdec` and `thmdisplay` (and all derived commands) display comments.

6.4 Special Declarations

The `declarepretheorem` command is used to indicate the intent of proving a theorem with a given name (this can be an operator).

The `declareopaque` and `declarenotopaque` commands can be used to enter and remove operators from the opacity list. Free variable operators can be declared opaque to block them from being declared with types. New restriction: declareopaque can now be used only to declare an infix initially; it is still possible to use declarenotopaque to remove opacity, but this is irreversible.

The `makescin` and `makescout` commands declare infixes or functions as having input or output (resp.) restricted to absolute types (strongly Cantorian sets). The user must supply a witnessing theorem. These declarations allow the stratification restrictions on terms involving these infixes to be weakened appropriately, without the use of explicit type retractions.

The `proveprogram` command can be used to bind a theorem to a constant or operator as a "functional program". The theorem must have a suitable form (the left side must be a substitution instance of a term which would be admissable as the left side of a definition of the constant or operator). The `deprogram` command removes the binding. The `seeprogram` (for one constant or operator) or `seeallprograms` commands can be used to view information about this list.

The `forget` command will eliminate declarations of theorems depending on or referring to a given theorem.

The definition commands are described above. A single definition can be viewed using the `seedef` command (it takes the name of the defined object as parameter and displays the defining theorem); all definitions can be seen using the `seealldefs` command.

6.5 Theorems

Entry of theorems into the theorem list is usually via the proof and definition commands. Axioms can be entered using the `axiom` command, which takes name, left side, and right side of the axiom as parameters, or using the `interaxiom` command, which interprets the selected subterm of the right side of the current equation as an equation (so it needs to be one!) and enters it as an axiom with a name supplied as a parameter.

One views a single theorem using the `thmdisplay` command. One can view all theorems using `showalltheorems` or `scanthms`. One can view the module associated with a theorem using `moddisplay`, or view all modules using `showallmodules`. One can view a theorem matching a given equation using `showmatchthm`. One can view a list of theorems applicable to the selected subterm (excluding theorems applicable to every term) using `showrelevantthms`. One can view all saved environments using `showsavedenvs`. One can view all axioms and definitions using `showallaxioms`.

6.6 Reaxiomatization and Redefinition

Theorems are stored with dependency information, a list of the names of axioms and definitions on which they depend. The dependency structure of a theory can be completely restructured.

The `makeanaxiom` command can force a theorem to become an axiom; the `proveanaxiom` command allows one to eliminate an axiom by exhibiting a proof of it from other axioms, with appropriate global dependency modifications. These tools can be used to support the use of lemmas before they are proved.

The `defineprimitive` command will convert a theorem into a definition of a primitive constant or operator if it has the right form. The `redefineconstant` or `redefineinfix` command will replace one definition of a constant or operator (resp.) by another, with appropriate dependency modifications. The `undefine` operator converts a definition to an axiom.

The elimination of unneeded dependencies on definitions (due to the defined object not being mentioned) should be handled automatically by the proof commands; this is the operational difference between a definition and an axiom. At the moment, however, performance problems with the function which removes unwanted dependencies on definitions make it necessary to remove all automatic applications; the user may apply the temporary `reallyremovedefdeps` command to effect such reductions of dependencies manually.

7 The Desktop Environment

This section discusses the management of theories and environments other than the current one, whether saved on the desktop or in files. It also discusses theory interpretations and their use in the theorem export facility which allows results proved in one theory to be exported to another.

7.1 Loading and Saving Theories

Theory files are saved with a fixed extension (currently `.wthy`). They are script files written using commands of the alternative interface invoked by the `walk` command which reconstruct all information in the theory.

There is an environment variable representing the name of the current theory, which can be read using the `theoryname` command. The `storeall` command sets the theory name to its parameter and stores the theory in a file with name obtained by adding the extension to the theory name. The `safesave` command saves a theory to the file whose name is indicated by the current theory name variable. The `load` command sets the theory name environment variable to its parameter and loads the appropriate theory file (automatically backing up the current theory on the desktop).

The `clear` command will clear the current theory; the `cleartheories` command will clear all theories from the desktop.

7.2 Proof Environment Management

Proof environments can be saved on the desktop while working in a particular theory. When a theory is saved on the desktop, all associated environments are saved with it (this is also true when theories are saved to files, since environments are stored on the theorem list).

The `saveenv` command allows one to save a proof environment, specifying a name/format to be associated with that environment. The `backupenv` command sets the name/format of the saved environment to the current name/format environment variable if set by the user or to `backup`. The `getenv` environment loads a saved environment; it will issue a warning if this environment was found only on the theorem list and not on the current desktop. The `getenv` command and most commands of the `start` family automatically invoke `backupenv`. An exception is if one is actually using `getenv` to retrieve the backup of the current environment!

`dropenv` will delete an environment (from the theorem list and the desktop) and `clearenvs` will eliminate all saved environments. `loadsavedenvs` allows one to load all saved environments in a theory file onto the desktop (one must hit return to load each such environment as in the thmdisplay family of commands); this can be useful if one wants to clear all such environments without loading them onto the desktop individually.

7.3 Theory Management

A theory is always associated with the name of a file.

Theories may be saved on the desktop using `backuptheory`. They may be retrieved using `gettheory`. The current theory may be cleared using `clear` and all theories may be cleared using `cleartheories`. `load` and `gettheory` automatically invoke `backuptheory`, unless one is using `gettheory` to retrieve the backup of the current theory.

A theory saved on the desktop is saved with all associated proof environments.

7.4 Views and Theorem Export

A `view` is an interpretation of a subset of a source theory in a target theory, implemented as a set of translations of names of constants and operators. Views reside in the source theory; they are saved in theory files.

Views are constructed using the `declareview` command, which creates a trivial view with the predeclared constants and operators interpreted as themselves. They can be edited using the `viewasin`, `viewasselfin`, and `dropfromview` commands.

Views are used to export theorems from the theory in which they reside (the source theory) to another theory (the target theory). The target theory needs to be saved on the desktop. The theorem export functions `exportthmlist` and `exportthm`, given a view, information about default prefixes to add to automatically generated names of constants and operators, a single theorem or list of theorems to be exported, and a target theory, will retrieve the target theory from the desktop, automatically check the validity of the view and extend it if necessary (the view must at lest include interpretations of all axioms and definitions on which the exported theorems depend, and often does not need to include more than this; the rest can usually be deduced), and generate analogous theorems in the target theory, guaranteed to be valid theorems of the target theory. The facility will automatically avoid name conflicts by adding additional prefixes to automatically generated names. If a tactic is exported, it will automatically determine what other theorems need to be exported and translate them as well; complex mutually recursive systems of tactics can be exported reliably. A reason to extend views beyond the minimal level of axioms and definitions is to avoid the copying of frequently used theorems found in many theories.

The theorem export commands can be finicky about defined notions in the source theory; these need to match defined notions in the target theory with analogous definitions. The redefinition facilities might be useful in avoiding problems with this.

The theorem export commands do not export module structure; they will export modules but they are unpacked completely.

8 Places where bugs are likely to be found

The theorem export facility is very complex and has not been tested under all possible perverse conditions; it is likely to be reliable but not perfectly reliable. Problems I am aware of involve the treatment of numerals!

The variable binding context is a new addition (originally, the bracket construction was used only for constant functions and there were no bound variables at all). I suspect it of having bugs, and I also suspect that I will find awkwardnesses in it which will require me to revise it somewhat.

9 Logical and Programming Style

9.1 Logical Style

The Mark2 prover has been affected from the outset by certain decisions of *logical style*. These are motivated by the theoretical origins of the project in untyped synthetic combinatory logic, motivated by Curry's program of untyped combinatory logic without bound variables on the one hand, and the untyped set theory "New Foundations" (*NF*) of Quine on the other.

9.1.1 The Main Points

Strictly equational: All reasoning under Mark2 is manipulation of equations using equations, though this may be automated in ways which can produce surprising effects. In particular, there is nothing precisely corresponding to the usual style of reasoning in propositional or predicate logic. Recently, we have been encouraged in this approach by the work of Gries, Schneider and Cohen on teaching an equational style of logic.

Avoidance of bound variables: Originally, Mark2 did not use bound variables at all. It is still the case that Mark2's variable binding facility is a sort of window dressing on top of the synthetic abstraction facility supported, for example, by the built-in theorem `RAISE`. The motivation for this comes out of the program of Curry, as well as work of Quine, Tarski and Givant, and others, on ways to formalize first-order logic while avoiding the use of variables. We were interested at the outset in the question of how much these theoretically effective but practically awkward approaches could be made more usable. A great deal of development was successfully carried out with no use of bound variables at all, and we feel that the early development of the prover benefited from the simplicity afforded by their absence, but our conclusion has been that bound variables have their place, both as having actual practical advantages and as making the prover's notation more familiar to hoped-for users. The current variable-binding system uses de Bruijn levels; users seem to be able to handle this.

Avoidance of types: This is a feature both of the systems of Curry and of Quine's set theory. In fact, the original logic of the prover (in an earlier version) was a synthetic combinatory logic equivalent to first-order logic on an infinite universe, which *did* have distinctions of sort. Moreover, the use of "New Foundations" was at first avoided, although the prover project grew out of an investigation of systems of combinatory logic equivalent to *NF*. Considerations of polymorphism between the two sorts of functions present in the original logic led us to abandon that logic (basically a first-order logic) and adopt a higher order logic equivalent to a consistent subset of *NF* (the consistency of full *NF* remains an open question). A special state of the prover implements a predicative version of *NF* which, while technically a higher-order logic, is actually weaker than first-order arithmetic; one regains full higher order logic upon issuing the `impredicative` command. The appearance of *NF* was not obvious; the only role it played in the earliest version of the prover using higher-order logic was to provide the stratification criterion for parameterized function definitions. The use of stratification had the desired immediate effect of providing complete polymorphism with respect to commonly used operations on functions.

We now have the additional feature of built-in type retractions onto "strongly Cantorian sets", which allow subversion of the stratification restrictions when one is working in "small" domains.

The original logic had a base sort containing all objects of the theory under consideration and a second and third sort consisting of functions on the objects of the base type and operations on these functions. All of these sorts were identified when the stratified higher-order logic was adopted. It cannot be said that the resulting identification of the domain of objects of a theory with domain of operators on those objects (only the type level ones, though) has been entirely convenient, but it has not proved wholly inconvenient, either. The lack of obligatory type notation is quite liberating (though it necessitates care). Type labelling can be introduced (use retractions as type labels) and it appears that typing schemes of this sort are natural objects for manipulation by the tactic language.

It should be noted that it is still not the case that all operators are first-class citizens of the theory in which they are used; only operators with trivial left and right type can be expected to correspond to functions of the theory.

9.1.2 The Logic of an Old Version

Some difficulties arise from each of these points of style. The solutions to these difficulties, and even the precise nature of the difficulties themselves, are best appreciated by examining the theory *EFT* ("external function theory") which was implemented in the original version of this theorem prover (the following is excerpted from an unpublished paper).

EFT is an equational theory. There are two (apparent) sorts, called objects and Functions. Object variables and constants will begin with lower-case letters; Function variables and constants will begin with upper-case letters. The "intended interpretation" is that the objects are the elements of some infinite set and the Functions are the functions from that set into itself (or a subset of the collection of such functions closed under certain operations). But see below for an alternate interpretation.

t and f are distinct atomic object constants (used to represent the truth values). If x and y are object terms, (x, y) is an object term, the ordered pair with projections x and y. If F is a Function term and x is an object term, $F[x]$ is an object term, the value of F at x or the result of application of F to x. Cond and Id are atomic Function terms. Id is intended to be the identity function; $\mathrm{Cond}[(x, y), (z, w)]$ is intended to be z if $x = y$, w otherwise. If x is an object term, $|x|$ is a Function term, the constant Function of x. If F and G are Function terms, (F, G) is a Function term, the product of F and G, and $F\langle G\rangle$ is a Function term, the composition of F and G. If F is a Function term, $F!$ is a Function, called the "Hilbert Function" of F. $F![y]$ is intended to be an object such that $F[F![y], y]$ is not t, if there is any such object; its value is a matter of indifference otherwise. We write $F[x, y]$, $F\langle G, H\rangle$, instead of $F[(x, y)]$, $F\langle (G, H)\rangle$, respectively. We define the n-tuple $(x_1, x_2, ..., x_n)$ inductively as $(x_1, (x_2, ..., x_n))$. Sentences of *EFT* are equations between object terms. The rules of *EFT* enable substitutions of equals for equals:

A. Reflexivity, symmetry, transitivity of equality.

B. If $a = c$, $b = d$ are theorems, $(a, b) = (c, d)$ is a theorem.

C. If $a = b$ is a theorem, $F[a] = F[b]$ is a theorem.

D. Uniform substitution of an object term for an object variable or of a Function term for a Function variable in a theorem yields a theorem.

Note that the rules do not directly permit substitutions of equals for equals where object terms appear as subterms of Function terms. The axioms are as follows:

(CONST) $|x|[y] = x$

(ID) $\mathrm{Id}[x] = x$

(PROD) $(F, G)[x] = (F[x], G[x])$

(COMP) $F\langle G\rangle [x] = F[G[x]]$

(PROJ1) $\text{Cond}[(x, x), (y, z)] = y$

(PROJ2) $\text{Cond}[(t, f), (y, z)] = z$

(DIST) $F[\text{Cond}[(x, y), (z, w)] = \text{Cond}[(x, y), (F[z], F[w])]$

(HYP) $\text{Cond}[(x, y), (F[x], z)] = \text{Cond}[(x, y), (F[y], z)]$

(HILBERT) $\text{Cond}[(F[F![y], y], t), (F[x, y], t)] = t$

It should be clear that the axioms are true in the intended interpretation (under the Axiom of Choice), and they should also serve to clarify the exact interpretations of the term constructions. If a version of the "intended interpretation" with a restricted class of functions interpreting the Functions of the theory is to be constructed, the axioms indicate the set of operations under which the restricted class of functions needs to be closed.

We have the following Abstraction Theorem:

Theorem: If s is an object term and x is an object variable which does not appear as a subterm of any Function subterm of s, there is a function term $(\lambda x)(s)$ such that x does not appear as a subterm of $(\lambda x)(s)$ and "$(\lambda x)(s)[x] = s$" is a theorem.

Proof: By induction on the structure of terms. If s is x, $(\lambda x)(s)$ is Id; if s is an atom a distinct from x, $(\lambda x)(s)$ is $|a|$. If s is of the form (u, v), $(\lambda x)(s) = ((\lambda x)(u), (\lambda x)(v))$. If s is of the form $U[v]$, U does not involve x and $(\lambda x)(s) = U\,\langle(\lambda x)(v)\rangle$.

$(\lambda xy)(s)$ such that "$(\lambda xy)(s)(x, y) = s$" is a theorem can be defined as $(\lambda z)(s_0)$, where z is a variable not appearing in s and s_0 is the result of replacing x with $\text{Cond}[(t, t), z]$ and y with $\text{Cond}[(t, f), z]$ wherever they appear in s. $(\lambda z)(\text{Cond}[(t, t), z])$ and $(\lambda z)(\text{Cond}[(t, f), z])$ are the projection Functions π_1 and π_2.

An interesting point about abstracts constructed following the proof of the Theorem is that they resemble the term from which they are abstracted in structure. This helps to make *EFT* an environment in which it is practical to avoid the use of bound variables.

9.1.3 Implementation of *EFT*

The precursor of Mark2 was EFTTP (*EFT* Theorem Prover) which had certain points of resemblance to Mark2 along with notable differences.

The notation of EFTTP followed the term structure of *EFT* exactly, with the addition of special term constructors for FUNCTORS (operations on Functions, the third sort) and the attachment of embedded theorems. The notation of EFTTP was very awkward; it did not support infix notation at all, for example (the current `?x + ?y` would be `+[x?,y?]` under EFTTP).

The avoidance of bound variables was made practical by the direct implementation of the Abstraction Theorem of the preceding section as a tactic. Parameterized tactics were not at that time available, but I will describe the `ABSTRACT` tactic in its modern form, as it appears in Mark2, rather than delving into the peculiar expedients of EFTTP. The tactic is invoked in the following format `(ABSTRACT @ term1) => term2`, and execution of the tactic yields a term `function_term @ term1`, in which, if `term2` can be expressed as a function of `term1` under appropriate constraints (such as stratification; obviously a different abstraction theorem is being implemented than in EFTTP!), the term `function_term` will not depend on `term1`. Moreover, as noted above, both in EFTTP and in Mark2, the algorithm implemented by `ABSTRACT` produces more or less readable terms parallel in structure to the terms from which they are abstracted, though not as readable as conventional λ-terms. The implementation of `ABSTRACT` is a subject for the section on programming style (NOTE TO BE REMOVED: be sure to note there why `RAISE` is needed; the problem of abstraction from operators). A tactic `REDUCE`, which needs no parameter, undoes the effect of `ABSTRACT`. The two tactics together allowed the automation of operations of global substitution under the prover, which made it possible to go quite far without any use of bound variables.

The Cond$[(x, y), (z, w)]$ construct of *EFT* allows the construction of case expressions. The analogous construction under Mark2 is `(?x = ?y) || (?z , ?w)` (the second set of parentheses would normally not appear). All reasoning under hypotheses is managed in EFTTP or Mark2 as equational reasoning about case expressions. The axioms (DIST) and especially (HYP) manage the system of hypothetical reasoning of *EFT*, which is adequate to the demands of propositional logic. In fact, a complete tautology checking tactic has been developed from the *EFT* axioms under both provers. Abstraction and reduction play an important role in the development of tactics which support this style of reasoning efficiently; the substitutions for arguments of functions in the axioms (DIST) and (HYP) can be converted to global substitutions in expressions of arbitrary form by the use of abstraction followed by reduction (the user never sees an abstraction term).

Reasoning about case expressions is implemented in a "hard-wired" way in the new hypothesis facility (see above).

The paper from which the description of the *EFT* logic in the previous subsection is drawn proves that *EFT* theories are precisely equivalent to first-order theories with infinite domain (the pairing forces the infinite domain).

The axiom (HILBERT) introduces an operator analogous to the Hilbert epsilon operator for defining quantifiers (HOL has a similar approach to defining quantifiers). A choice operator could be used in a similar way in Mark2's logic; if it is desired that the Axiom of Choice not be assumed, this can be made possible by declaring the choice operator as "opaque" to abstraction.

9.1.4 The Logical Preamble

The prover automatically loads the declarations of certain logical operators and axioms governing these operators. Most of these are derived from axioms of *EFT*. They can be examined by starting the prover and looking at the axioms that are present before any theory is loaded.

9.2 Programming Style

The development of the tactic language has largely been driven by the requirements of the `ABSTRACT` and (to a lesser extent) `REDUCE` tactics. An outline of the development of some simple example programs culminating in the development of limited (predicative) abstraction and reduction tactics and an introduction to some of their applications seems natural as an introduction to programming in the tactic language. These tactics are no longer used in practice, with the introduction of variable binding and the `BIND` and `EVAL` built-in tactics, but tactic programming remains useful.

9.2.1 Basic Control Structures

Sequencing: The term `thm_5 => thm_4 => thm_3 => thm_2 => thm1 => term`, when "executed" will apply the theorems or tactics numbered 1 through 5, in that order, to the original term.

Alternation: The term `thm_5 =>> thm_4 =>> thm_3 =>> thm_2 =>> thm_1 => term`, when "executed", will apply the first of the theorems or tactics numbered 1 through 5 (as above, in that order, working from the inside out) which applies to the term. No more than one of the tactics will be applied. In principle, the sequencing construction could be used for this as well, if one were certain that applying one of the tactics would not give a form to which one of the later tactics in the list might unexpectedly apply.

Conditionals: All conditional execution in the tactic language depends on the fact that a theorem is not applied if it does not match its target. The termination of recursive tactics, for example, depends on this.

Guarded commands using `*>` and `<*`: The term `(thm_n *> guard_n) =>> ... (thm_1 *> guard_1) => term` will apply the first of the theorems `guard_i` which apply to the term, and then will further apply `thm_i`. An advantage of this is that `thm_i` might be constructed using sequencing on a very general term, and so might apply under much more general circumstances than those specified by applicability of `guard_i`; before this construction was available, it would have been necessary to construct a new theorem with the effect of each composite `thm_i *> guard_i`.

9.2.2 Recursion

The real power of the tactic language arises from the possibility of proving recursive or mutually recursive tactics. A simple example is an expansion tactic for algebra:

Suppose we have the following theorems available:

```
COMM:  ?x * ?y = ?y * ?x

DIST:  ?x * (?y + ?z) = (?x * ?y) + (?x * ?z)

COMMDIST:  (?x + ?y) * ?z = (?x * ?z) + (?y * ?z)

    (* COMMDIST is obviously provable from axioms COMM and DIST *)
```

We declare a pretheorem `X` and prove the following theorems (tactics):

```
X1:  ?x + ?y = (X => ?x) + (X => ?y)

X2:  ?x * ?y = X1 => COMMDIST =>> DIST => (X => ?x) * (X => ?y)
```

The tactic `X1` is used to expand sums: it can be summarized by "expand each term".

The tactic `X2` is used to expand products: it can be summarized by "expand each factor, then apply the first appropriate of distributivity from the left and the right, then expand each term if the result is a sum".

After proving these, the theorem `X` is anticlimactic:

```
X:  ?x = X1 =>> X2 => ?x
```

All that this tells us to do is to apply whichever of `X1` and `X2` might be appropriate. Note that `X` applies successfully to any term whatever; if one wanted to restrict circumstances under which `X` would be applied, one might want to use the `*>` or `<*` operator.

9.2.3 Abstraction and Reduction Algorithms

In this section, verbatim text from prover output is presented. The reader should remember that default precedence under Mark2 is equal for all operators, and that all operators associate to the right.

We present a limited abstraction tactic. This tactic `ABSTRACT` has the form

```
ABSTRACT @ ?x:

?y =

(ABSCONST @ ?x) =>> (ABSINFIX @ ?x)
=>> (ABSCOMP @ ?x) =>> (ABSID @ ?x) => ?y
```

The tactic takes as a parameter the term with respect to which we will abstract, and applies a tactic taken from a list of alternatives (each of which inherits the parameter.

```
ABSID @ ?x:

?x =

Id @ ?x
```

The only thing which distinguishes the `ABSID` subtactic from the converse of the defining axiom of the identity function `Id` is the presence of the parameter, which ensures that only the term from which we are abstracting will have this subtactic applied to it.

```
ABSCOMP @ ?x:

?f @ ?y =

COMP <= ?f @ (ABSTRACT @ ?x) => ?y

COMP:

(?f @@ ?g) @ ?x =

?f @ ?g @ ?x
```

We display the `ABSCOMP` subtactic and the defining axiom `COMP` of the function composition infix `@@` on which it depends. The recursive application of `ABSTRACT` should convert `?y` to something of the form `?Y @ ?x`, and application of the converse form of `COMP` to `?f @ ?Y @ ?x` will give `(?f @@ ?Y) @ ?x`. Note that it is assumed that `?x` does not appear as a subterm of the function `?f`; this restriction is essentially the same as the restriction on abstraction in *EFT*. It is possible to write more powerful abstraction tactics, but this one has proved adequate for most applications we have worked on.

```
ABSINFIX @ ?z:
```

```
?x ^+ ?y =

RAISE => ((ABSTRACT @ ?z) => ?x)
^+ (ABSTRACT @ ?z) => ?y
```

The built-in theorem RAISE is described elsewhere in this documentation. If ^+ is assumed to have trivial relative types, RAISE has the effect of the theorem RAISE0: (?f @ ?x) ^+ (?g @ ?x) = (?f :^+ ?g) @ ?x. The theorem has more complicated effects on infixes which have nontrivial relative types. The reason why the theorem RAISE is needed is that operators (at least, operators with nontrivial type information) are not first-class objects in the logic of Mark2, and abstraction with respect to operators presents logical difficulties. The theorem RAISE0 above can now be stated as an axiom or proved as a theorem using RAISE, but even this kind of abstraction was not possible until we permitted the declaration of infix variables with fixed type information.

The effect of the recursive applications of ABSTRACT should be to convert the whole term to the form RAISE => (?X @ ?z) ^+ (?Y @ ?z), and application of RAISE will convert this to the form (?X :^+ ?Y) @ ?z.

The three subtactics above handle the cases where we see the term from which we are abstracting by itself, or where we see a function application or an infix term. In all other cases, we use a constant function as our abstraction:

```
ABSCONST @ ?x:

?y =

[?y] @ ?x
```

This tactic differs from the converse of the defining axiom for constant functions only in having a parameter. Its effect is to express ?y as a function of ?x in a trivial way. The alternative control structure is very useful here, since this subtactic applies to any term at all! The abstraction tactics written before alternative rule infixes were available applied this subtactic first, then looked at the result and reversed it where it could be recognized that the other cases applied; these algorithms were a little harder to understand!

The execution of the term (ABSTRACT @ ?x) => ?t always has the effect of producing a term of the form ?T @ ?x; for a wide class of terms t the term T will not contain any instance of the term ?x. The term matching ?x does not have to be a variable!

We present a reduction algorithm which "undoes" the abstraction algorithm above.

```
REDUCE:

?x =
((LEFT @ REDUCE) *> (RIGHT @ REDUCE) *> RAISE)
<<= ((RIGHT @ REDUCE) *> COMP) =>> CONST =>> Id
=> ?x
```

This algorithm has simpler recursion than the ABSTRACT algorithm, but it involves the use of more sophisticated control structures. The theorems Id and CONST (not exhibited) have the effect of applying identity and constant functions respectively. The theorem COMP is exhibited above (the definition of function composition). The effect of the *> operator is to cause (in the first instance) the theorem RIGHT @ REDUCE to be applied as well when COMP is applicable (this is the "guarded command" format described above). We examine the parameterized theorem RIGHT:

```
RIGHT @ ?th:

?x ^+ ?y =

?x ^+ ?th => ?y
```

This theorem takes a theorem ?th as a parameter, and applies it to the right subterm of the term to which the parameterized theorem is applied. This is useful when the subterm to which this theorem is to be applied does not yet exist in the term to which the parameterized theorem is applied (in this case, the subterm in question is to be created by the application of COMP).

A segment of output of the tactic debugger is exhibited to illustrate how this works:

```
((LEFT @ REDUCE) *> (RIGHT @ REDUCE) *> RAISE)
<<= ((RIGHT @ REDUCE) *> COMP) =>> CONST =>> Id
=> (?f @@ Id) @ ?x

(RIGHT @ REDUCE) => ?f @ Id @ ?x

?f @ REDUCE => Id @ ?x
```

The parameterized theorem LEFT used above is exactly analogous in form to RIGHT, except that it operates on left subterms. A related theorem is the theorem $, (used below) exhibiting the possibility of theorems whose names are operators. This theorem has the effect of applying the theorem which is its argument in its "converse" form:

```
 $?th:

?x =

?th <= ?x
```

In connection with proving theorems like LEFT, RIGHT, and $, it is useful to note that the rule introduction commands will accept variables as theorems!

The effect of the complex theorem ((LEFT @ REDUCE) *> (RIGHT @ REDUCE) *> RAISE), applied in the reverse sense, which is the last alternative in REDUCE is to apply the converse of RAISE (if it is applicable) and follow this by reducing the left and right subterms of the resulting term.

The use of guarded command format eliminates the necessity in earlier versions of special theorems for each of the separate cases.

As an example of the application of abstraction and reduction algorithms, we present a tactic which implements the axiom (HYP) of *EFT* in a very powerful form:

```
PIVOT:

(?a = ?b) || ?x , ?y =

(RIGHT @ LEFT @ REDUCE) => HYP => (?a = ?b)
|| ((ABSTRACT @ ?a) => ?x) , ?y
```

The effect of this tactic is to substitute the term matching ?b for the term matching ?a throughout the term matching ?x. The effect of abstraction is to convert ?x to something like ?X @ ?a. The theorem HYP from the logical preamble can then be applied:

```
HYP:

(?a = ?b) || (?f @ ?a) , ?c =

(?a = ?b) || (?f @ ?b) , ?c
```

and the reduction algorithm (directed to the appropriate subterm) cleans up. The user never sees an abstraction term:

```
PIVOT => (?a = ?b) || (?a & ?a) , ?b

val it = () : unit
- execute();


{(?a = ?b) || (?b & ?b) , ?b}
```

The use of theorems like PIVOT may be largely superseded by the new facility for reasoning under hypotheses (see above), but it is still an instructive programming example.

We exhibit abstraction and reduction theorems which use the new variable binding capabilities of Mark2: these algorithms actually use the synthetic abstraction and reduction algorithms in conjunction with the built-in theorem VALUE (documented here) and the axiom BETA also given (note the use of the theorem $ described above):

```
BETA:

[?f @ ?1] =

?f

BETAABSTRACT @ ?x:

?y =

(LEFT @ VALUE @ [REDUCE]) => (LEFT @  $BETA)
=> (ABSTRACT @ ?x) => ?y

BETAREDUCE:

?f @ ?x =

REDUCE
=> (BETA => (VALUE @ [ABSTRACT @ ?1]) => ?f) @ ?x
```

When working with variable binding contexts, it is useful to remember that there are strong restrictions on the conditions under which free variables of Mark2 can match terms containing bound variables.

It is useful to be aware that the new theorems EVAL and BIND implement abstraction and reduction directly for functions expressed by variable binding contexts; use of the bound-variable-free abstraction features of the prover is now entirely optional.

10 Command Reference

This is a description of the theorem prover on an abstract level organized around data types. The order is top-down.

This is an older layer of documentation and its consistency with the remarks above may not be complete; it also may not refer to all commands referenced above. I will be working on remedying this.

10.1 Command Line

The interface to the prover is a command line. Each command consists of a command name followed by a number of arguments. The number and type of arguments is determined by the specific command and is never variable. The types of arguments which occur are strings, integers, and terms of the internal language. In the ML interface, string and term arguments are both handled as strings and so enclosed in double quotes; term arguments need special delimiters, as they may include significant spaces. There is a variant command line interface with single letter commands and arguments separated by tabs in the current ML version; it is invoked by the "walk" command. Command names in the variant interface can be seen by typing "h" in that interface. Another command line interface, very similar to the ML interface, is invoked by the `noml` command; this uses the same file parser used by the `script` command.

The standard interface of the current prover is actually the ML interpreter; it is useful to be aware that ML requires all arguments except integers to be enclosed in double quotes, uses tilde for integer negation, requires an argument () for commands with no parameters, and requires that each line end with a semicolon. See the tutorial for examples.

10.2 List Scanning Commands

This subsection describes a family of commands for viewing various kinds of lists in the prover. Each of these commands puts one in an environment with the following one-letter commands:

l "left" – go to the item with alphabetically previous label.

r "right" – go to the item with alphabetically next label.

Enter view this item.

q quit.

h (or any other key) "help" – see a list of commands.

The commands available include `scandecs` (declarations), `scanprograms` (functional program bindings), `scantheorems` (theorems), `scanenvs` (saved proof environments), `scantheories` (theories on desktop).

10.3 Command Abbreviations

These are abbreviations for names of commonly used commands found below.

```
(* /////////// ml declarations to change the names of commands //////////// *)
fun ri x = ruleintro x;
fun rri x = revruleintro x;
fun ari x = altruleintro x;
fun arri x = altrevruleintro x;
fun s x = start x;
fun p x = prove x;
fun rp x = reprove x;
fun ex() = execute();
fun di x = declareinfix x;
fun du x = declareunary x;
fun a x = axiom x;
fun dpt x = declarepretheorem x;
fun sp x n = setprecedence x n;
fun td x = thmdisplay x;
fun wb() = workback();
fun ae x = autoedit x;
fun tri x = targetruleintro x;
fun smt x = showmatchthm x;
fun srt() = showrelevantthms();
fun dti s t = declaretypeinfo s t;
fun uti s = detypeinfo s;
fun sat()  = showalltheorems();
```

10.4 Desktop

A desktop consists of several workspaces on the desktop and one current workspace. The workspaces on the desktop may include a version of the current workspace. Each workspace is associated with a theory (there can be no more than one workspace associated with each theory); the name of a workspace (unless it is "scratch") will be the name of a saved theory file.

The new commands `versiondate` and `script` are treated as desktop commands more or less by default, as are some recently introduced commands for controlling behavoir of scripts.

Desktop commands include

versiondate: Displays the date of the version of Mark2 being used.

script <string>: This command has the effect of the ML `use` command, but adds the extension `.mk2` to its argument. Files with this extension are to be expected to be ML proof scripts. The `script` command no longer uses the ML interpreter; a new file parser has been writtem.

setpause, setnopause: `setpause` causes the prover to pause whenever it issues an error message. This is useful when running proof scripts. `setnopause` reverses this effect.

bequiet, speakup: `bequiet` suppresses all output usually seen from proof scripts, except error messages. `speakup` reverses this effect, which is necessary if normal prover function is to be restored.

noml: Invokes a new interface which uses the same file parser as the `script` command. Commands in this interface have the same format as in the ML interface, as long as the new file parser is familiar with the command. The () (type unit) parameter to commands which don't have a real parameter is optional with this interface; the null `quit` command exits this interface (and must end scripts to be run by the `script` command).

demoline <string>: Used to make comments with pauses in demo scripts.

initializecounter: Sets the new variable counter to zero. Useful in scripts, as otherwise names of new variables prove unpredictable.

storeall <string>: Save the current workspace to a theory file whose name is the argument with an extension added (currently ".wthy").

safesave: Stores the current workspace to a file; determines the argument from the name of the workspace.

load <string>: Load a saved theory from the theory file indicated by the argument, backing up the current theory onto the desktop.

clear: Set the current theory to "scratch" (nothing but the logic prelude).

clearpredicative: As `clear`, with the additional "predicativity" restrictions on abstraction. Of technical interest only.

gettheory <string>: The argument is the name of a theory. Get this theory off the desktop (backing up the current theory unless the name of the current theory itself is the argument).

backuptheory: Save the current theory on the desktop.

exportthm, exportthmlist (parameters not indicated): Commands for theorem export. Parameters of exportthmlist are a view name, a prefix to add to constants automatically generated, a prefix to add to operators automatically generated, a list of theorems to be exported, and the name of the target theory (which must be on the desktop). Parameters of exportthm omit the prefix for operators and replace the list of theorems with a single theorem. Theorem export is likely to generate many warnings of repeated constant declarations.

10.5 Workspace

A workspace consists of a number of "saved" proof environments, all associated with the same theory, whose name is the name of the workspace, and a current proof environment (one of the "saved" environments may be an older version of the current one).

Workspace commands include

theoryname See the name of the current theory.

start <term>: Create a proof environment, with name a default, and with the term under consideration given by the argument. Any existing proof environment is backed up.

startfor <term> <term>: As "start", except that the name of the proof environment is specified by the first term argument.

getleftside *or* **getrightside <string>:** Create a new proof environment with starting term the appropriate side of the theorem indicated by the argument and name set to the "format" of that theorem.

autoedit <string>: Creates a proof environment with name, left side of equation and right side of equation determined by the theorem whose name is the argument.

getenv <term>: Get the proof environment named by the argument, backing up the current one unless the current one is the one being retrieved.

saveenv <term>: Save the current proof environment with the indicated name.

backupenv: Save the current proof environment under its own name or as "backup" if its name is the default.

10.6 Proof Environment and Theorem

The "proof environment" and "theorem" data types are closely related and will be discussed together here.

A proof environment has five ingredients: a name (which can usually be thought of as a string but must technically be regarded as a term, because it may be parsed to determine the "format" of the theorem to be proved (no longer true; name a format are now distinct)), the left side of the equation being proved, the right side of the equation being proved, a list of names of axioms on which the theorem to be proved depends, and, finally, a list of booleans which represents the position of a specific subterm of the right side of the equation to be proved.

A theorem has a name, a "format" (a term which determines with what arguments the theorem should be invoked), a left side, and right side, a list of dependencies, and a list of theorems used by the module system.

It appears that the name and format ought to be distinct elements of proof environments (this is now the case). Dependencies are expected to be expanded to include dependencies on definitions and on programs; this will improve reaxiomatization and theorem export facilities.

There are no commands specifically manipulating the theorem type (theorems appear as components of theories, for which see below).

Proof environment commands include the following (for commands creating proof environments, see the workspace subsection; for command which actually prove theorems, see the theory subsection):

envname See the name of the current environment.

look *or* **lookhere** *or* **lookback:** "look" displays the right side of the equation (with the current subterm in braces) above and the current subterm below. Invoked automatically by many commands. "Lookhere" shows the current subterm only; "lookback" shows the left side of the equation.

lookhyp <integer>: Displays the hypothesis indexed by the integer (see discussion of the hypothesis facility above).

lookhyps: Displays all hypotheses.

seedeps: See the list of names of axioms on which the current theorem depends.

startover *or* **starthere** *or* **workback:** Manipulations of general equation structure. "Startover" sets both sides of the equation to the left side and clears dependencies, and "starthere" does the same with the right side. "Workback" interchanges the two sides of the equation and leaves dependencies unaffected.

left *or* **right** *or* **up** *or* **top:** Basic "movement commands". These cause the current subterm to be reset to its "left child", "right child", "parent", or the whole right side of the equation, respectively. "Constant function" terms have only one "child", which is treated as both left and right child; if one "moves" left or right from an atomic term, one gets an error condition, but it is currently necessary to move up or to the top to escape this condition (the list of booleans does get set to the non-existent position!)

upto *or* **downtoleft** *or* **downtoright <term>:** Sophisticated movement commands; they move as indicated (up, or down to left or right) until a term matching the argument is reached. If no such term is found, an error is reported; the position one ends up in depends on the command.

New movement commands: Commands `uptols <theorem>`, `uptors <theorem>`, `dlls <theorem>`, `dlrs <theorem>`, `drrs <theorem>`, and `dlrs <theorem>` enable movement to places where theorems can be applied. Commands beginning with `upto-` implement "upto"; commands beginning with `dr-`

and `dl-` implement "downtoleft" and "downtoright"; the suffix indicates "left side" or "right side".

ruleintro *or* **revruleintro <term>:** Introduce an application of a theorem or its converse (indicated by use of the connectives "=>" or "<=" respectively in the displayed result). The theorem may have parameters, thus the argument must be a term. The dependencies of this theorem are added.

matchruleintro *or* **targetruleintro <term>:** Introduce an embedded theorem or theorem converse which matches a given equation (the first command) or will convert the current subterm to a given target term (the second command).

altrule: If the current subterm is an infix term with infix "=>" or "<=", convert the infix to =>>, <<= respectively (or the reverse). This converts embedded theorems and converse theorems to the form appropriate for lists of alternatives no more than one of which should be applied.

droprule: Drop an embedded theorem from the top level of the current subterm.

delay *or* **undelay:** Introduce or eliminate the special prefix "#" which converts infixes with "functional programs" associated to "inert" form.

lazy *or* **unlazy:** Introduce or eliminate the full laziness prefix **##**.

execute *or* **steps** *or* **onestep:** Invoke the interpreter (resp. debugger) for the internal programming language. In general terms, execute causes all embedded theorems to be applied following a depth-first strategy; "steps" simulates "execute" step by step with some limited parallelism. More parallel versions Execute and Steps of these commands are expected to be implemented. The steps command goes into a loop which is broken by hitting "q"; it stops automatically if the term stabilizes. If you break out of any of the tactic interpreters using Control-C, you should issue the "top" command to reset environment variables.

applythm *or* **applyconvthm <term>:** Apply the indicated theorem or converse theorem (possibly with parameters); does not apply any new embedded theorems produced by this application. Useful for editing.

applyenv *or* **applyconvenv <term>:** As above, except that saved environments are used.

assign *or* **assigninto <term> <term>:** Carries out global assignments to variables; can affect both sides of the equation under construction. The first argument is a variable term. The second argument is any term.

In the case of "assign", the second argument is substituted for the first throughout the left and right sides of the current equation. In the case of "assigninto", the left and right sides of the current equation are substituted for the variable first argument in the second argument to obtain the new left and right sides.

assignit <term>: Equivalent to the assign command with the variable first argument given on the command line and the second argument the current subterm.

10.7 Theory

A theory consists of a list of theorems, a list of constant declarations, and some auxiliary lists of declarations. Theories can be saved to and loaded from files.

The lists which comprise a theory at this time (other than lists used by the parser which will be discussed below under Term) are

CONSTANTS: Master list of constant declarations. Contains all identifiers and infixes; for infixes, reports the relative types of their arguments. Now contains comments.

THEOREMS: Master list of theorems. For the structure of a theorem, see above.

PRETHEOREMS: List of identifiers or infixes which will eventually be proved as theorems (needed for recursive theorem definitions).

DEFINITIONS: List of defined identifiers and infixes with their defining theorems.

PROGRAMS: List of bindings of identifiers and infixes to theorems used for simulation of functional programming.

OPAQUE: List of operators declared as opaque to abstraction.

These could reasonably be collapsed into one or two master lists. Additional lists used by the parser referenced under Term below are also an essential part of a theory.

Theory commands include the following:

declareconstant <string>: Declare an identifier as a constant.

declaretypedinfix <integer> <integer> <string>: Declare an infix. The string is its form; the integers are the "relative types" of its left and right arguments.

declareinfix <string>: Declare a "flat" infix (with both relative types 0)

declareunary <string>: Declare a unary prefix (its default left argument will be `defaultprefix`).

newcomment <string> <string>: Enter new comments (the second string) on the declared object named by the first string (any older comments will be superseded). These will be displayed by declaration and theorem display commands.

appendcomment <string> <string>: As `newcomment`, except that comments are appended to existing comments.

makeunary <string>: Give an existing infix the default left argument `defaultprefix`.

declarepretheorem <string>: Declare an identifier (or infix) as a prospective theorem (so that it can be introduced as an embedded theorem; useful in cases of recursion).

declareopaque <string>: Declare an infix as opaque to abstraction. Used to declare new infix; an existing infix cannot be made opaque.

makescin *or* **makescout <string> <string>:** These commands allow the user to declare the function or infix denoted by the first string argument to have input or output (resp.) restricted to absolute types (strongly Cantorian sets). The second string argument must be the name of a theorem witnessing this fact.

proveprogram <string> <string>: Bind to the constant or infix named by the first argument the theorem named by the second as a "functional program".

deprogram <string>: Remove any program bound to the constant or infix named by the argument.

pushtheorem *or* **pushtheorem2 <string> <string>:** Hide the theorem named by the first string in the module associated with the theorem named by the second string. The hidden theorem is only visible when the parent theorem is executing, but its name remains reserved. The first version of the command posts a comment to the declaration of the hidden theorem automatically; the second does not.

poptheorem *or* **poptheorem2 <string> <string>:** Reverses the effect of `pushtheorem`. `poptheorem` posts a comment automatically to the declaration of the no longer hidden object; `poptheorem2` does not.

axiom <string> <term> <term>: Creates an axiom with name the first string argument, left side the second argument and right side the third argument. An axiom can be recognized by its list of axiom dependencies consisting of its own name. Axioms cannot take parameters, so the first argument does not need to be a term.

interaxiom <string>: If the current subterm is an equation, makes it an axiom.

prove <term>: Causes the equation expressed in the current proof environment to be recorded as a theorem with name and "format" determined by the term argument (the argument will be either the theorem name alone or the theorem name with a list of parameters). The theorem to be proved cannot already be a theorem, though it can be a pretheorem.

reprove <term>: Allows one to change an existing theorem; the only restriction enforced is that the dependencies of the theorem cannot get stronger. Axioms and definitions cannot be reproven.

autoprove *or* **autoreprove:** As "prove" or "reprove", except that the format of the theorem proved is the name of the current workspace. Especially useful in a workspace created by "getleftside", "getrightside" or "autoedit".

impredicative: This command allows definitions of more general form and allows the built-in RAISE theorem to work on non-flat infixes. The default state of the prover is now impredicative.

defineconstant <term> <term>: Introduce a theorem defining a constant, with the same name as that constant. The constant cannot previously have been declared. The first term is a constant name or a constant applied to a list or lists of parameters, which is the left side of the intended theorem; the second term is the right side of the intended theorem. The structure of the definition is checked. Definitions are recorded in dependency lists like axioms.

definetypedinfix <string> <integer> <integer> <term> <term>: As defineconstant, except that an infix with specified left and right types is defined. The first string argument is the name of the theorem to be added (since a solitary infix is not a theorem name). The fourth and fifth arguments correspond to the arguments of defineconstant.

defineinfix <string> <term> <term>: Defines "flat" infixes (with both relative types 0). The first argument is the theorem to be added; the second and third correspond to the arguments of defineconstant.

defineitconstant, defineittypedinfix, defineitinfix: have arguments as above, except for the last, which is replaced by the current subterm.

defineopaque <string> <term> <term>: Defines opaque infixes. The arguments are as for "definetypedinfix", with the omission of relative types. Stratification restrictions on the definition are removed, but the use of the defined function in definitions of anything non-opaque or in variable binding contexts is strongly restricted.

makeanaxiom <string>: Makes the theorem indicated by the argument an axiom.

proveanaxiom <string>: If the current proof environment contains a proof of an axiom named by the argument, make that axiom a theorem with the appropriate dependencies, making the appropriate adjustments of dependencies through the whole theory.

undefine <string>: Convert the definition of the constant or operator named by the parameter to an axiom.

defineprimitive <string>: If the theorem named by the argument is of the appropriate form to serve as the definition of an undefined notion, make it so.

redefineconstant <string>: If the current proof environment could serve as a definition of a constant now defined, make it so, with global dependency modifications as appropriate. The argument is the new name for the old definition theorem.

redefineinfix <string>: If the current proof environment could serve as a definition of an operator now defined, make it so, with global dependency modifications as appropriate. The argument is the name of the new defining theorem.

Note: The preceding commands support reaxiomatization of theories. This facility can also be used to support the use of as yet unproved lemmas.

thmdisplay <string>: Displays a theorem. Displays comments. Displays names of module components (hidden theorems) but not their contents.

showalltheorems: Displays all theorems (in a loop from which any key other than Enter breaks out).

moddisplay <string>: Displays a module (a theorem and all theorems hidden in its module).

showallmodules: Displays all modules (in a loop from which any key other than Enter breaks out).

showrelevantthms: Displays all theorems which match (or whose converses match) the current subterm nontrivially (i.e., it does not display theorems which match all terms).

showmatchthm <term>: Displays a theorem (if there are any) which justifies the equation given as the argument.

showdec <string>: Displays a constant or infix declaration. it shows relative types and default left arguments in the case of infixes usable as prefixes. Displays comments.

showalldecs: Shows all declarations (in a loop as with showalltheorems).

10.8 Term Syntax and Display

We describe the syntax of the input language. Tokens are either *identifiers* (made of alphanumeric characters and the characters "_" and "?") or *infixes* (made up of special characters, excluding parentheses, quotes, braces, and brackets).

Identifiers have two special subsets: numerals (all characters numeric) and variables (starting with "?"). Other identifiers are called "constants". Infixes starting with a caret are infix variables and an initial colon is an operator on infixes which may be iterated.

Constants and infixes other than variables must be declared to be used. Predeclared constants include "RAISE" (a built-in theorem), "true" and "false". Built-in infixes include "=", "," (pairing), "@" (function application), "=>", "=>>", "<=", and "<<=" (theorem embedding infixes), "+!", "*!", "-!", "/!", "%!", "=!", and "<!" (operators of the built-in unsigned integer arithmetic).

Single identifiers of whatever class are terms. Any term enclosed in parentheses is a term. A term preceded by an infix (here used as a unary prefix) is a term. An infix flanked by terms is a term. A term enclosed in brackets is a term (a constant function whose value is the enclosed term).

In the current version, each infix used as a prefix has associated with it a default left argument (selected by the user). In other words, all unary prefix terms are abbreviations for binary infix terms. If an infix for which a default left argument has not been declared is used as a prefix, a bogus value is used which signals an error condition to the prover. It should be noted that the "left" movement command *can* for this reason be used with prefix terms. The declareunary command above should be used for prefixes which are not also to be used as infixes; the declareprefix command below can be used to set desired behavior where overloading is intended.

The default operator precedence assigns to all infixes and prefixes the same precedence and associates to the right as far as possible. The user can assign integer precedences to prefixes/infixes (and this information can be stored in theories); higher precedences bind more tightly; even precedence associates to the right and odd to the left.

Two lists are used by the parser and are actually components of the theory type: PREFIXINFO contains infixes with default left arguments and PRECEDENCEINFO contains precedence information. Saved theories are actually stored in default precedence in the current version, and the precedence list is loaded at the end of the process. Parsing is done in a first pass independent of declaration checking; the parser will parse and display terms which the prover will tell you contain undeclared constants or infixes.

There is some pretty-printing built into the prover's display function; some indenting is done in a way which is intended to suggest the structure of terms. There are commands to set the right margin of the display and to set the "depth" of the display.

Commands which relate to term display include:

declareprefix <string> <string>: Declares the default left argument of a prefix (the first argument) to be a constant indicated by the second argument.

setprecedence <string> <integer>: Sets precedence of an infix (the first argument) to an integer value (the second argument). Even precedence operators associate to the right and odd precedence operators to the left.

declaretypeinfo <string> <string>: Assigns an implicit strongly Cantorian type to a variable (the first parameter, without leading ?). The effect is to cause the type infix : to function as a unary operator on that variable with invisible left argument determined by the second parameter. This information is currently not stored in theory files.

detypeinfo <string>: Removes implicit type from a variable.

showprecs: Shows a table of precedences (including explicit indication of left or right association).

clearprecs: Restores all precedences to the default.

setline <integer>: Set the right margin for term displays.

setdepth <integer>: Limit the depth to which terms are displayed. Negative values cause all terms to be displayed; this is the default.

nodepth: Removes any limit on depth of display (sets the relevant variable to a negative value).

10.9 The Logic of the Definition Facility

The commands for the definition facility are given in the theory subsection. Here we discuss the theory of the definition facility briefly.

A term is said to be a "parameter list" if it is built from variables using the operation "," of pairing. A term is said to be in "definition format" if it is either a constant or a term in definition format applied to a parameter list (using the infix "@" of function application). A term is said to be in "infix definition format" if it is the result of applying a unary prefix to a parameter list, the result of applying a binary prefix to two parameter lists, or the result of applying a term in infix definition format to a parameter list using the "@" connective. The infix cannot be a variable or colon-initial infix.

A term in either definition format has a "head", the sole constant or infix other than "@" or "," which appears.

A "definition" is a theorem whose left side is in definition format or infix definition format and which serves as the definition of the head (whether constant or infix) of the left side of the equation.

A definition needs to satisfy a number of technical requirements. The head (the notion being defined) cannot have been declared at the time the definition is made. This avoids circular definition neatly. The right side must have no variables other than those which occur in the left side and must contain only declared constants and infixes.

To understand the further requirements, it is necessary to explain the relative type system of the prover. The types are indexed by integers (including negative integers). Type $n + 1$ can be thought of as the type of functions from type n to type n. Terms do not have fixed types, but subterms of a term are assigned types relative to the whole term. The relative types of left and right arguments of an infix subterm are determined by adding the left and right types associated with that infix by its declaration to the type of the whole term; the relative type of the immediate subterm of a constant function subterm (of its value) is one less than that of the parent term.

The relevant additional constraint on the form of definitions is that each variable in the theorem (considering it as a single equation term) occurs with the same relative type everywhere in the term. In addition, no variable should occur in a term with an "opaque" infix, and no variable should occur in the left argument of a function application infix in a theory started with the `clearpredicative` command unless the "impredicative" command has been issued in that theory (this "predicative" state used to be the default state of the prover – old theory files may need the impredicative command).

Definitions do not have any axiom dependencies. They are projected to have definition dependencies soon so that definitions can be reconfigured in the same way that axioms can now be reconfigured.

The logic of the definition facility is the "higher-order" logic of the prover; in conjunction with the logic prelude loaded as part of every theory, it gives an implicit logical strength equivalent to Russell's Theory of Types.

10.10 List of Error Messages

This may not be entirely up to date.

`Division by zero!` Message from built in arithmetic with obvious meaning.

`At beginning`, `At end` Messages from the `scan` command sent when you try to read before the first entry or after the last entry in a tree.

`<return> to view,l to go left, r to go right, q to quit` Sent by the `scan` command when an inappropriate letter is entered.

`No negative precedence` The `setprecedence` command does not permit negative precedence to be set.

`Errors found in displayed term` This message is issued by the term display command when it encounters the error term `Constant ""`.

`Warning:  adding leading parenthesis` This warning is issued by the parser when it is forced to add a leading parenthesis to make sense of a term.

`Subterm error` This message is issued by the movement commands when one attempts to move to a subterm of an atomic term.

`Ill-formed hypothesis, Ill-formed case expression` These messages are issued by the declaration checking function when it encounters inappropraite uses of `|-|` or case expressions without a pair of alternatives as the second argument, respectively.

`Illegal opacity declaration` Issued by the commands for declaring opaque infixes when they do not succeed. Possible reasons include ill-formed infix names, conflicts with previous declarations.

`Undeclared constants or embedded theorems present` Issued by a declaration checking function of the prover when it finds a term unacceptable, usually because it contains undeclared constants or operators or was so ill-formed as to be impossible to parse.

`Meaningless bound variable or unstratified abstraction error` Issued by a declaration checking function when it finds errors in stratification or in indexing of deBruijn levels (variable binding) in a term.

`Illegal or repeated constant declaration of, Illegal or repeated infix declaration of` (it will supply the declared object). Issued by the declaration commands when a declaration is illegal or such an object has already been declared. These messages will normally be issued many times during theorem export.

`Name already in use` Issued by `declareopaque` when an attempt is made to redeclare a preexisting constant.

`Not an infix` Issued by `declareopaque` when a non-infix parameter is provided.

`Constant is a numeral` Issued by `showdec` when one attempts to show declaration of a numeral (numerals are "predeclared").

`Opaque infix variable` Issued by `showdec` when one attempts to show declaration of an infix variable declared as opaque.

`Constant not found` Issued by `showdec` when no constant with the given name is found.

`Theorem not found` Issued by `thmdisplay` when there is no such theorem. Also used by `reprove`.

`Built-in operation` Issued by `thmdisplay` when one attempts to display a "theorem" which is actually a hard-wired tactic.

`No definition found` Issued by `seedef` when no such defined notion is found.

`Program not found` Issued by `seeprogram` when no program is bound to the given infix or function (or when the infix or function itself is not understood).

`Not an equation!` Issued by commands which match theorems when the object presented to match with theorems is not an equation.

`No such theorem` Issued by `showmatchthm` when no matching theorem is found. Also issued by `applythm` and `applyconvthm` when they do not recognize their arguments.

`Illegal theorem declaration` Issued by `axiom` when parsing or declaration problems prevent an axiom from being accepted.

`Stratification, opacity or impredicativity error in definition` Issued by the `defineconstant` command when stratification or related considerations prevent a definition from being accepted.

`Declaration or syntax error in definition` Issued by the `defineconstant` command when undeclared objects appear or there are parsing problems; this includes the case of circular definitions.

`Stratification or opacity error in infix definition` Issued by infix definition commands when stratification-related considerations cause a definition to be rejected.

`Warning: automatically declaring default left argument of defined prefix` Issued by infix definition commands when an operator is implicitly defined as unary.

`Declaration or syntax error in infix declaration` Issued by infix definition commands when undeclared objects or syntax errors appear in a definition. This includes attempts at circular definition.

`Argument is not defined appropriately` Issued by the `undefine` command when the object supplied to be "undefined" was not defined in the first place.

`Inappropriate argument` Issued by the `defineprimitive` command when it is given an unsuitable theorem or a non-theorem.

`Definition format error` Issued by `defineprimitive` when it is given a theorem which does not have the form of a definition.

`Not a theorem` Issued by `makeanaxiom` when the argument is not a theorem.

`Not applicable to definitions`, `Not applicable to built-in theorems` Issued by `makeanaxiom` when argument is a definition or a built-in theorem.

`Illegal pretheorem declaration` Issued by `declarepretheorem` when syntax or declaration errors cause it to reject a declaration.

`Theorem is not of right form` Issued by `proveprogram` when theorem is not of the correct form to be bound to the given function or infix.

`Impossible error in program declaration` This error message should never be issued by `proveprogram`, barring bugs.

`Undeclared object error or program already present` Issued by `proveprogram` when object is undeclared or already has a program bound to it.

`Deprogram what???!` Issued by `deprogram` when it does not recognize its argument.

`No program to delete or change` Issued by `deprogram` when there is no program to remove.

`At top already` Warning issued by the up command when one attempts to go up from the top of a parse tree.

`No such hypothesis` Issued by `lookhyp` when its argument is inappropriate.

`Bad environment name or corrupt environment` Issued by `saveenv` or `backupenv` when an attempt is made to save or back up a corrupt environment or to save an environment with an ill-formed name. Warning: the message will appear when a perfectly good new term is started if the previous environment was corrupt, due to the automatic attempt to back up the bad environment.

`Reconstructing environment from theory` Remark made by `getenv` when it is loading an environment from a saved theory into the local data structure which stores environments.

`Environment not found` Issued by `getenv` when it finds no environment named by the argument.

`Cannot drop backup` Issued by `dropenv` when it is told to drop the current backup environment.

`No match found` Issued by `upto` and related commands when no matching term is found to which to move.

`Invalid argument` Issued by `upto` and related commands when the argument doesn't pass a declaration check (or doesn't parse).

`What theorem??!!` Issued by commands like `uptols` when argument is not recognized as a theorem.

`Built in theorem not supported by apply(conv)thm` Issued by `applythm` and `applyconvthm` when they encounter built-in theorems which they do not support.

`No such environment` Issued by `applyenv`, `applyconvenv` when argument is not recognized.

`Corrupt environment` Issued by `applyenv`, `applyconvenv` when rerenced environment is corrupt (this actually should not happen).

`Undeclared constants in theorem to be proved` Issued by `prove` when theorem does not pass a declaration check. This may signal stratification errors as well as declaration errors.

`Conflict with existing theorem name (or illegal name or format)` Issued by `prove` when it objects to the proposed name or format of a theorem.

`Environment name not appropriate` Issued when an attempt is made to use `autoprove` when there is not a user-defined environment name.

`Cannot modify an axiom` , `Cannot modify a definition` Issued by `reprove` when an attempt is made to modify an axiom or definition.

`Introduces additional dependencies` Issued by `reprove` when it rejects a change in a theorem because it strengthens dependencies.

`This command can only be issued at top of term` Issued by `interaxiom` when an attempt is made to interpret a proper subterm as an axiom.

`Current subterm is not of correct form` Issued by `interaxiom` when the current subterm is not an equation.

`Target is not an axiom, is a definition, or does not match environment` The all-purpose error message of `proveanaxiom`.

`Incorrect new name for old definition` Issued by `redefineconstant` when declaration collisions or syntax make the proposed new name for the old definition inappropriate.

`Source is not a suitable definition` Issued by `redefineconstant` when it does not like the proposed new definition.

`Declaration or syntax error in redefinition` Issued by `redefineconstant` probably only when the proposed new definition would be circular.

`Inappropriate arguments` Issued by `redefineinfix` for a variety of offenses against declarations.

`Stratification or opacity error in infix redefinition` Issued by `redefineinfix` when stratification or related considerations cause new definition to be rejected.

`Declaration or syntax error in infix redeclaration` Issued by `redefineinfix` probably only when definition would be circular.

`Illegal substitution attempted` Issued by `assign` and relatives when bad substitutions are attempted.

`No rule to drop!` Issued by `droprule` when it is invoked inappropriately.

`No rule to toggle` Issued by `altrule` when it is invoked inappropriately.

`Theorem or pretheorem not found` Issued by `ruleintro` when it does not recognize its argument as a theorem.

`Ill-formed theorem name` Issued by `ruleintro` when it cannot parse its argument to find a theorem name.

`Declaration error` Issued by `ruleintro` when its argument contains undeclared constants or cannot be parsed at all.

`No matching theorem found` Issued by `matchruleintro` or `targetruleintro` when it cannot find a matching theorem to introduce.

`Bad arithmetic evaluation` Internal error – I doubt that the prover can actually issue it to a user.

`No such theory to get` Issued by `gettheory` when it does not understand its argument.

`Won't drop backup of current theory` Issued by `droptheory` when it declines to drop the current backup.

`No such theory found to drop` Issued by `droptheory` when it does not understand its argument.

`View of that name already exists` Issued by `declareview` in event of name collision.

`No such constant to project` Issued by `viewasin` when it does not understand its local declaration argument.

`Cannot export infix without its definition` Issued by `viewasin` when an attempt is made to export a defined infix without its definition.

`Already found in view` Issued by `viewasin` when there is already a binding in the view.

`Such an item is not found in such a view` The all-purpose errormessage of `dropfromview`.

`No such view to drop` Issued when `dropaview` does not understand its argument.

`No such view` Used by `viewasin`, `viewofin`, `showview`, or a theorem export command when it does not recognize its view argument.

`No such item in view` Used by `viewofin` when it does not recognize its binding argument.

`Bad deps in export list` An error in theorem export. The theorem(s) to be exported depend on axioms not found in the view to be used.

`Constant in view is not declared` An error in theorem export. A constant in the view is not declared.

`Constant definition matching error,Defined infix matching error` An error in theorem export. A definition does not match.

`Missing declarations in target` An error in theorem export. Something the view leads one to expect is not present in the target theory.

`Infix typing error` An error in theorem export. The types of an infix do not match correctly.

`No such remote theory` Theorem export command does not recognize the target theory.

`Construction of export list failed` An error in theorem export. Export list could not be constructed.

`Predicativity error in export` An error in theorem export. Source theory is impredicative and target theory is predicative.

`Opacity error in export` An error in theorem export. Something in target theory is opaque and its counterpart in source is not.

`Theory match failure` An error in theorem export. Source and target theories do not match as claimed by the view.

`Export aborted; theory restored` Remark issued on backtracking from a failed theorem export.

`Illegal character command` Message from the `walk` interface when it does not recognize a command.

`Unknown or inappropriate theorem(s)` Message from the `pushtheorem` or `poptheorem` commands when given inappropriate arguments; the `forget` command sends a similar message (in the singular).

11 Tutorial

The MARK2 theorem prover is an upgrade of the theorem prover EFTTP, an equational prover implementing first-order logic, to allow the implementation of higher order logic.

The prover is equational: all theorems are equations, and the rules of inference supported directly are exactly the basic rules of equational reasoning that we all learn (implicitly at least) in algebra.

The prover makes no use of bound variables; all variables in theorems are free and may be regarded as implicitly universally quantified. Features of the input language make it possible to use synthetic abstraction while retaining readibility.

The prover incorporates a programming language. This was originally developed to allow the easy implementation of "tactics", procedures for the automatic implementation of many steps of reasoning without user intervention. Tactics or programs look to the system like equational theorems, and are stored in theories along with theorems of the normal sort; we think that the way in which this is achieved has some interest. The fact that the language has its own tactic-writing facility makes it much less dependent on the languages in which it is written (two dialects of ML) than was originally expected. We are currently working on a C++ implementation, which will allow more precise control over details of memory allocation and other aspects of execution.

The prover is interactive. The user enters a term and manipulates it, possibly with the help of automatic tactics, until he or she obtains the desired final form of the term. The user can then prove a theorem equating the initial and final forms of the term, which acquires the same status as the axioms and previously proved theorems of the system. Tactics are developed in the same way. Facilities are also provided for "debugging" tactics (programs).

The last couple of paragraphs of this introduction are technical and can safely be skipped.

The semantics of the prover are based on a system of relative typing ("stratification") analogous to that which underlies Quine's set theory "New Foundations". It is closely related to a streamlined version of the type system of the typed λ-calculus widely used in computer science, and can often be regarded as implementing this streamlined system of types with polymorphism. The type system is completely invisible to an unsophisticated user!

The types in this "streamlined" type system are labelled by non-negative integers n, with each type $n+1 = n \rightarrow n$ (functions from type n to type n) and each type n identified with type $n \times n$ (pairs of objects of type n).

The consistency difficulties of "New Foundations" do not apply here; this system is related to the version *NFU* of Quine's system which is known to be consistent. For a technical discussion of the theory of stratified λ-calculus, see our preprint "Untyped λ-calculus with relative typing".

11.1 Building the Prover if You are Somewhere Else

Run the source code under SML/NJ (Standard ML of New Jersey) then issue the command

```
exportML "sml_prover";
```

to create an executable file (actually a version of SML/NJ with the prover commands pre-loaded).

11.2 The Platform

MARK2 is implemented in SML/NJ (Standard ML of New Jersey) and Caml Light (another dialect of ML, which runs on PC's!)

To start the SML version from UNIX on onyx, first type

```
sml_prover
```

You will get some verbiage and an SML prompt.

The fact that we are in ML has two effects. One is that you will always issue commands at the ML prompt (a hyphen). Another is that ML will send you some messages (it reports the type of its output to you after each command). I don't know a way to suppress this.

The direct effect it has on you has to do with the syntax of command lines. Each prover function is actually an ML function. Those which have arguments take string arguments, which must be enclosed in double quotes (there are a couple of commands which take integer parameters; this will be pointed out). ML insists that those which do not have arguments take a dummy argument, written () and known affectionately as "unit". ML command lines must end with semicolons. In the Caml Light version, all command lines must end with *two* semicolons.

Sample command lines:

```
showalltheorems();  (* a command with no parameters *)

axiom "COMM" "?x+?y" "?y+?x";  (* a command with three string
arguments *)
```

If you goof up in ML, ML will send you error messages. There is no easy way to damage your theorem prover session by outraging ML; just ignore commands which lead to ML errors and try again.

To abort a line and start over, type Control-C. To leave ML altogether, type Control-D. If Control-C is typed to break out of one of the command interpreters (execute, steps, or onestep) be sure to use the top command to reset environment variables.

To save your work in MARK2 in a new "theory file", type

```
storeall "<name of theory>";
```

Theory files have the extension ".thy3", added automatically by the "storeall" command. To open an existing theory file, issue the command

```
load "<name of theory>";  (* don't include the .thy3 extension! *)
```

A lot of stuff will happen; you may have to wait a little while if the theory file is large.

If you have introduced a theory name by using either of the above commands, the command

```
safesave();
```

will automatically save your work to the correct file.

11.3 The Input Language

Terms to be manipulated by the prover as representing objects in mathematical theories are entered by the user and displayed by the prover in the MARK2 "input language".

The syntax of the input language is very simple. There are two types of tokens. *Identifiers* are strings of letters, numbers, question marks and underscores. Identifiers which begin with question marks are used as variables; all others are used as constants (when so declared by the user). *Infixes* (also used as prefix operators) are strings of special characters (not including brackets, braces, or parentheses). Those which begin with carets (shift-6) are variable infixes; infixes which begin with exclamation points are the product of an infix construction to be described below. Infixes which a user declares will not begin with exclamation points or carets.

A term is either of the form <identifier>, the form [<term>] (such terms are called "constant function terms", for reasons which will become evident), the form (<infix> <term>) (such terms are called "prefix terms") or the form (<term> <infix> <term>) (such terms are called "infix terms").

It is not necessary to enter all parentheses in the above term forms. The rule followed is the old APL rule; all operators associate to the *right* as far as possible. Spaces are not significant, except where they are required to separate adjacent infixes (only occurs when an infix is followed immediately by a prefix). The prover automatically displays terms with the least possible use of parentheses. Variables do not need to be declared.

The best way to get to know the input language is to enter into dialogue with MARK2. It is first necessary to be aware that all constant identifiers and infixes used must be declared. It is also necessary to use declarations to tell the prover that a binary infix can be used as a prefix.

Try entering

```
start "?x+?y";
```

You should get an error message telling you that an undeclared constant has been used. You should also see two slightly reformatted copies of the term you have entered. The "start" command always displays two copies of the term entered, for reasons which will become evident below.

Now type

```
declareinfix "+";
```

and repeat the "start" command from above. MARK2 should no longer complain.

Now try the following three commands:

```
start "(?x+?y)+?z";
start "?x+(?y+?z)";
start "?x+?y+?z";
```

Feel free to try more complex terms; explore what is meant by "associating to the right". Try including a constant like "a" (not beginning with a question mark) and see what MARK2 says. Try putting in numerals like "0" or "12"; you will find that MARK2 will not give us trouble with declarations in this case.

Now we look into prefixes. Enter the command

```
declareunary "-";
```

You will discover that this makes possible the use of "–" as both an infix and a prefix operator. Explore this using the "start" command. Especially notice the difference between

```
start "?x+-?y";  (* There is no infix +- *)
start "?x+ -?y";  (*  The space is necessary *)
```

The form of the command for declaring constants like "a" is

```
declareconstant "a";
```

Declare some literal constants and mix them into expressions introduced with the "start" command. Try declaring some new infixes and prefixes.

Try outraging the declaration system by declaring constants or infixes which have already been declared, or by declaring constant identifiers with special characters, "constant" identifiers beginning with question marks, infixes with letters in them, and so forth. MARK2 has a large library of error messages :-)

Just for the sake of completeness, we explain the mechanics behind prefixes. A prefix expression is always understood by the system as an abbreviation for an infix expression. Each infix has associated with it a "default value" in such a way that the infix term (<default-value1> <infix1> <term1>) (where the appropriate default value depends on the infix) is the intended meaning of the prefix term (<infix1> <term1>) If the former term is entered, the latter term will be displayed. The default value associated with any infix normally is an error value which will cause MARK2 to complain; the default value associated with infixes declared using "declareunary" is `defaultprefix` (all infixes without exception can legally be used as infixes; there is no way to declare an infix term which can only be used as a prefix).

The command which associates a given default value with an infix is

```
declareprefix "<infix>" "<default-value>";
```

To test out this system, try the following commands:

```
declareconstant "true";
declareinfix "~";
start "true~?x";
start "~?x'';
declareprefix "~" "true";
start "true~?x";
start "~?x";
```

Try entering terms like "0−?x" or complex terms with embedded subterms of that sort.

Save your theory using the command

```
storeall "tutorial";
```

To start again, use

```
load "tutorial";  (* in the same directory you were in when you stored
it! *)
```

and it can be saved then using

```
safesave();
```

11.4 Navigation within Terms

Enter the term $?a+?b$ using the "start" command. Now type the command

```
right();
```

The lower occurrence of $?a+?b$ should be replaced with its right subterm $?b$. Now type

```
up();    (* you should return to the whole term *)
left();  (* you should see the left subterm *)
left();   (*  you should get an error message *)
top();  (* takes you back to the top level *)
```

Enter a more complex term like $(?a+?b)+?c+?d$ and use the movement commands "right", "left", "up", and "top" until you have a feeling for how they work. To recover from the *Subterm error* condition you will have encountered above, either (possibly repeated) use of "up" or a single use of "top" will work.

The official description of these phenomena follows. The display which you have seen following each application of the "start" command shows two terms: the top term is the complete term currently being considered, while the lower term is the "current subterm" of that term.

The "left" or "right" command causes the "current subterm" to become the left or right subterm of the erstwhile current subterm. The left or right subterm of an infix term is defined in the obvious way. The left and right subterms of a constant function term $[x]$ are both the subterm x. Atomic identifiers (constant or variable) have no left or right subterms, and so an error condition is reported. Prefix terms do have left subterms (the "default values" associated with their prefixes): try entering

```
start "-?x";

left();
```

The "up" command causes the current subterm to become the smallest subterm of the complete term which properly contains the erstwhile current subterm; if one is at the top level already, an error message is sent. The "top" command causes the current subterm to become the complete term under consideration.

The underlying metaphor views terms as (upside-down) trees, with the complete term as the root (at the top!) and each subterm having its immediate subterms growing as branches below it.

The more complex movement commands "upto", "downtoleft", and "downtoright" enable one to move to subterms remote from ones current "position"; they will be described under the description of individual commands in the Appendix.

If one is in doubt about how to parse a term of the input language, the movement commands provide a way of exploring the structure of a term. For example, the left subterm of $?a+?b+?c+?d$ is $?a$ and the right subterm is $?b+?c+?d$, which might be a useful discovery if one were uncertain about our associativity convention. Look at the structure of the term $?a*?b+?c$ (after declaring an infix *); it is not what you might intuitively expect, since there is no precedence of operators in the language.

A command which is often useful is the command

```
look();
```

which allows one to see the complete term and the current subterm again; some prover commands leave you not able to see these.

11.5 Starting to Prove Something

So far we can write down terms, but neither we nor the prover knows anything about what they mean. We begin to remedy this.

We describe the fundamental structure of a prover session. The user enters a term using the "start" command. We call this the initial term. The user performs a series of manipulations of this term (sometimes of the initial term as well), guaranteed to preserve the truth of the equation between the initial term and the current term. The user finally issues a command, usually

```
prove "<name-of-theorem>";
```

which records the theorem <initial-term> = <current-term> under the name <name-of-theorem>. The manipulations allowed in the intermediate steps are most often applications of equational theorems, and the new theorem just proved will be added to the current "theory" (library of theorems) and will be usable in subsequent proofs.

Of course, not all theorems in the library can have been proven; some are introduced initially as axioms or as definitions. Definitions will be discussed below; we now show how to enter an axiom. Issue the following command.

```
axiom "COMM" "?x+?y" "?y+?x";
```

A display produced by the prover will follow. The prover now knows that the infix operator "+" is commutative.

The format for an axiom declaration should be clear from the above example; the "axiom" command has three parameters, the first being the name of the axiom, the second being the left side of the equation being asserted as an axiom, and the third being the right side of the equation.

Our assertion that the prover "knows" that addition is commutative requires some support. Try out the following sequence of commands.

```
start "?x+?y+?z";
ruleintro "COMM";
```

The current term now looks like this:

```
COMM => ?x + ?y + ?z
```

The infix => is a predeclared operator. The value of $?x => ?y$ is the same as that of $?y$, with the annotation that one intends to apply the theorem $?x$ to the term $?y$. Such terms are called "embedded theorem applications".

Now issue the command

```
execute();
```

which carries out all embedded theorem applications in the current subterm. You should now see the term

```
(?y + ?z) + ?x
```

which is the result of applying the commutative law of addition to the original term.

There is another opportunity to apply the commutative law here; this can be realized by the sequence of commands

```
left();
ruleintro "COMM";
execute();
```

which should leave you with the term

```
(?z + ?y) + ?x
```

You should now be able to see why the movement commands are provided. Next, use "ruleintro" to propose application of the commutative law to both the complete term and the subterm at the same time, then use the "top" command to return to the complete term, and only then issue the "execute" command. Both applications of the commutative law will be carried out at once, returning the term to its original shape. The form of the term before you issue the "execute" should be

```
COMM => (COMM => ?z + ?y) + ?x
```

As the last exercise illustrates, the commutative law of addition is completely symmetrical in its effect. We introduce another axiom, by the command

```
axiom "ASSOC" "(?x+?y)+?z" "?x+?y+?z";
```

with which we will explore further features of theorem application.

Issue the following sequence of commands:

```
start "?a+?b+?c";
ruleintro "ASSOC";  execute();  (* you can write more than one command
     on a line! *)
```

Did you expect something to happen? (It is useful to note that an embedded theorem which does not apply to its target is simply removed by the "execute" command).

The difficulty is that the theorem application represented by the infix => always represents the result of attempting to match the target term with the *left* side of the theorem; the command which works in this situation is

```
revruleintro "ASSOC";
```

upon which we see

```
ASSOC <= ?a + ?b + ?c
```

An infix expression $?x$ <=$?y$ has the value of $?y$ and signals the intention of applying the theorem $?x$ in reverse; that is, attempting to match the term $?y$ with the *right* side of the equation. Now, the "execute" command gives the desired result

```
(?a + ?b) + ?c
```

A useful exercise is to take the term $(?a+?b)+?c+?d$, or a similar complex term, and try applying the associative law to it in each direction. A useful command to know about is

```
startover();
```

which restores the initial term.

We now prove a theorem, the theorem $?x+?y+?z = ?z+?y+?x$, which we will call "REVERSE". Enter this sequence of commands:

```
start "?x+?y+?z";
revruleintro "ASSOC";  execute();
ruleintro "COMM";  execute();
right();  ruleintro "COMM";  execute();
prove "REVERSE";  (* the new theorem should now be displayed *)

startover();
ruleintro "REVERSE";  execute();  (* the new theorem can be applied
     like the old ones *)
```

Type

```
thmdisplay "REVERSE";
```

to get another look at your theorem (this is a useful command to know anyway). Note especially the list of strings at the bottom of the display; this indicates the axioms on which "REVERSE" depends.

Another command of the same genre is

```
showalltheorems();  (* after typing this, hit enter repeatedly *)
```

Each time you hit enter, a theorem will be displayed; it will go through the entire list. Depending on the context you are working in, you may see some extra theorems you don't know anything about. You can use any key other than Enter to break out.

An exercise which one might attempt now is to introduce the axioms of commutativity and associativity for multiplication (represented by *) as well as the distributive law of multiplication over addition, and prove the usual FOIL rule $(?x+?y) * (?z+?w) = (?x*?z) + (?x*?w) + (?y*?z) + (?y*?w)$. You might decide that you wanted to prove some additional theorems to make this easier (for example, if you stated the distributive law as $?x * (?y+?z) = (?x*?y) + (?x*?z)$, you might want to prove the other form, $(?x+?y)*?z = (?x*?z) + (?y*?z)$, for use in the proof of the FOIL theorem.

A subtlety of theorem use can be illustrated if we introduce the axiom "INV": $?a + -?a = 0$. Issue the commands

```
start "0";
revruleintro "INV";  (* after declaring the axiom! *)
execute();
```

If you declared the axiom exactly as described above, you will get

```
?a_1 + -?a_1
```

The numerical suffixes (which may take a different value than 1) are added to make collisions with variables that you are already using unlikely. We can assign a specific value to "$???a$" by using the "assign" command:

```
assign "?a_1" "3";
```

whereupon we have

```
3 + -3
```

In this most common use of the "assign" command, it is expected that the variable to which the assignment is made is *new*; it is important to note that if an assignment is made to a variable which appears in the initial term (the term you introduced with "start" at the beginning of your session) that the variable will be replaced in the initial term as well as in the current term; this is obviously necessary if assignments to variables are to preserve an equation between the initial term and the current term.

We use the command "lookback" which allows us to look at the initial term without returning to it in the following example:

```
start "?x+?y";
ruleintro "COMM";  execute();
assign "?x" "?a+?b";
lookback();  (* the initial term will have changed! *)
look();      (* the current term is unchanged by the lookback command
     *)
startover(); (* note that this does not undo the assignment *)
```

With the "ruleintro", "execute", "assign", and "prove" commands, you have the basic tools needed to do laborious step by step proofs by hand in MARK2. If you prove interesting theorems, be sure to save your work (using the "storeall" command if you started from scratch, or the "safesave" command if you are working in an existing theory which you got with the "load" command, as described above).

11.6 Definitions

MARK2 provides a facility for defining new constants:

```
defineconstant "Four" "2+2";
```

The effect of this command is to create a theorem with the same name "Four" as the constant being declared. The constant "Four" must not have been declared previously! Notice that the list of axioms on which this new theorem depends is empty; it is a "freebie". One cannot define a constant which depends on variables.

It is also possible to define new infix operators. For example, we can define a new infix "**" with the meaning "sum of the squares of its arguments" as follows:

```
defineinfix "SUMOFSQUARES" "?x**?y"  "(?x*?x)+?y*?y";
```

Note that the user needs to supply a name for the theorem encoding the definition.

It is also possible to define functions in the same general style as a mathematical definition like

$$f(x) = x^2 + x.$$

The equivalent MARK2 definition looks like this (try it):

```
defineconstant "SampleFunction@?x" "(?x*?x)+?x";
```

The infix @ is a predeclared infix used to represent function application: the mathematical term $f(x)$ would be written $?f@?x$. Another predeclared infix is the comma, which is used to represent the ordered pair construction, and can also be used in parameterized definitions:

```
defineconstant "Add@?x,?y" "?x+?y";
```

defines a function of two variables encoding the addition operation.

It is possible to define higher-order functions and to define infix terms via their function values. Examples follow:

```
defineconstant "(Comp@?f,?g)@?x" "?f@?g@?x"  (* composition of
     functions *)
defineinfix "ADDFNS" "(?f++?g)@?x" "(?f@?x)+?g@?x"
```

There are restrictions on definitions having to do with the underlying type system of the prover. An example of a definition which will not succeed is

```
defineconstant "Diag@?x" "?x@?x";
```

We will not go into the details at this point; generally, if one defines functions which make sense mathematically, one will not run afoul of the type system (but don't try to define general terms from λ-calculus without reading about the details of the type system below).

The definition system is secure against circular definition or the presence of unacknowledged parameters.

The definition of any concept creates a theorem; one uses that theorem with "ruleintro" to eliminate the defined concept and that theorem with "revruleintro" to introduce the concept. Try out the following session:

```
defineconstant "Square@?x" "?x*?x";
start "(?a+?b)*(?a+?b)";
revruleintro "Square"; execute();
ruleintro "Square"; execute();
```

11.7 Simple Programming and Debugging

Suppose in this subsection that we have available the axioms "COMM": $?x+?y = ?y+?x$ and "ZERO": $0+?x = ?x+0$.

An obvious theorem to prove (prove it) is "COMMZERO": $?x+0 = ?x$.

But we usually feel that "ZERO" and "COMMZERO" are not really separate theorems; it would be nice to be able to express that adding zero on either side of a term leaves it unaffected. We will prove a theorem "EITHERZERO" which has this effect. The session proceeds as follows:

```
start "?x+?y";
ruleintro "ZERO";
ruleintro "COMMZERO";
prove "EITHERZERO1";
```

The theorem "EITHERZERO1" looks like this:

```
?x + ?y =
COMMZERO => ZERO => ?x + ?y
```

Try applying it to terms $?x+0$ and $0+?x$. It seems to have the desired effect.

The reason that this works has to do with the way the "execute" command works. The command applies all embedded theorems, including ones introduced in the course of execution, always applying the innermost ones first. You can get a look at how it works by using the command

```
steps();
```

instead of "execute", then hitting return repeatedly. It will trace the execution of the various embedded theorems for you. The "steps" command can be used as a debugging tool. Use "q" to break out (it will stop automatically if the term stabilizes).

Now try applying the theorem "EITHERZERO1" to 0+?x + 0. You should get ?x; the tactic got a little overenthusiastic. You can figure out why on your own or look at the execution using "steps". This behaviour can be corrected:

```
start "?x+?y";
ruleintro "COMM";
ruleintro "COMMZERO";
altrule();  (* see what happens? *)
prove "EITHERZERO";  (* note difference in theorem names *)
```

The new theorem looks like this:

```
?x + ?y =

COMMZERO =>> ZERO => ?x + ?y
```

The =>> infix represents an embedded theorem which will be applied only if the theorem which was to be applied immediately before was inapplicable to its target; so, if "ZERO" is applied, "COMMZERO" will not be applied; applying "EITHERZERO" to 0+?x + 0 will give ?x + 0 instead of ?x. An example of its use: apply the first of the four theorems $x1$, $x2$, $x3$, and $x4$ which can be applied to a term y, use $x4$ =>> $x3$ =>> $x2$ =>> $x1$ => y (the last one must be a => or nothing will happen at all!); this avoids the possibility that $x1$ might apply to y, and, say, $x3$ apply as well to the result, which might happen if => were used throughout. To apply a theorem in reverse only on failure of previous theorem, use <<=, which has the same relation to <= that =>> has to =>. The ability to construct a list of alternative theorems to be applied while being certain that the outcome will not be that several of them will be applied in sequence helps to make behaviour of complex theorems of this kind more predictable.

With “EITHERZERO1” and “EITHERZERO”, we are already doing programming (recall that theorems which function as programs are called “tactics”). But we can do much more impressive things.

We now develop a tactic which applies “ZERO” aggressively, removing every addition of zero that it can find!

```
declarepretheorem "ZEROES";
```

The “declarepretheorem” command declares the identifier “ZEROES” and tells the prover that a theorem by this name will be forthcoming. Normally, the “prove” command takes care of this automatically, but this will not work in this case.

```
start "?x+?y";
right(); ruleintro "ZEROES"; up();
left(); ruleintro "ZEROES"; up();
```

The prover takes our word for it that there will soon be a theorem called “ZEROES”.

```
ruleintro "EITHERZERO";
prove "ZEROES";
```

Something is very fishy here. The theorem “ZEROES”, which looks like this:

```
?x + ?y =
EITHERZERO => (ZEROES => ?x) + ZEROES => ?y
```

seems to be defined in terms of itself (which was why it was necessary to declare it)! Try applying this theorem to a complex sum with lots of parentheses and zeroes; it should hunt down and eliminate all the zeroes. Watch it at work with “steps” (use “q” to break out of “steps”).

The reason that the recursion terminates is that embedded theorems simply disappear when applied to terms to which they are inapplicable; notice if you use “steps” that the applications of “ZEROES” to atomic terms do not produce further occurrences of “ZEROES”. It is definitely possible to write recursive tactics which will *not* terminate, by the way; to break out of such a process, use

Control-C and then use the top command to reset environment variables (and then use "steps" to see what went wrong).

The possibility of defining tactics in terms of themselves gives us the "looping" (actually recursive) control structure; the fact that theorems or tactics fail where they do not apply gives us a limited "conditional" control structure (refined by the use of =>> and <<= to construct lists of alternative theorems to be applied), which proves to be sufficient to break out of recursions. Elaborate mutual recursions are possible, and are useful in practice.

The tactic language built into MARK2 is itself a programming language. And programs written in this language are treated by the prover as equational theorems, stored with other theorems in saved theories on an equal footing.

An exercise would be to use the experience gained in writing the theorem "FOIL" assigned above to write a tactic "EXPAND" which aggressively expands terms written using addition and multiplication as far as possible, applying the distributive law (in either form) wherever possible. Another exercise is to write a theorem "SUPERASSOC" which will eliminate all parentheses from a complicated sum, by applying the theorem "ASSOC" until it is no longer possible to do so.

11.8 Appendix: Reference for Individual Commands

All arguments of commands must be enclosed in double quotes (except for certain integer arguments); commands without arguments must be supplied with the dummy argument (). All command lines end with a semicolon. Double quotes and () will not be shown in the individual command references.

11.8.1 Interfaces

walk The "walk" command invokes a new user interface independent of ML (this is probably its only virtue!). Commands in the "walk" interface are single characters. Arguments (which may have spaces in them) are separated by tabs. The character versions of commands are noted below under each command. To quit the walk interface type "q" (to quit and save), "Q" (to quit without saving). For help type "h"; a list of character commands is displayed.

noml The "noml" command implements an interface independent of ML for which command format is almost the same; the unit tpe parameter required by ML for commands with no real parameters is optional at the noml interface. One does not get messages from ML, but one also does not have a prompt, which may be distracting. `noml` uses the same file parser as the `script` command.

11.8.2 Theory Loading and Saving Commands

load "load file-name" causes the theory stored in file-name.wthy to be loaded into the workspace. If a theory has already been loaded, it is backed up on the desktop. Load sets the "current theory" environment variable to "file-name" as well. (walk command "G").

safesave The safesave command has no arguments. It calls the "storeall" command with argument the "current theory". The "walk" interface version is "S".

storeall "storeall file-name" stores the current theory to file-name.wthy. It also sets the "current theory" environment variable to file-name. (walk command "F").

clear The clear command erases the current theory and loads preambles (axioms always used). It does not affect other theories on the desktop. It sets the "current theory" to "scratch" (walk command "C").

clearpredicative As "clear", and imposes predicativity restrictions on abstraction. Of technical interest only.

versiondate This command (no parameter) displays the date of the version of the software being used.

script "script file-name" has the effect of the ML command "use file-name.mk2"; files with the extension `.mk2` are to be recognizable as proof scripts. `script` now uses a file parser independent of ML, also exploited by the `noml` alternative interface command.

The next two commands control prover output in ways that are sometimes convenient when running proof scripts.

setpause, setnopause: `setpause` causes the prover to pause whenever it issues an error message. This is useful when running proof scripts. `setnopause` reverses this effect.

bequiet, speakup: `bequiet` suppresses all output usually seen from proof scripts, except error messages. `speakup` reverses this effect, which is necessary if normal prover function is to be restored.

11.8.3 Environment Starting and other Modification Commands

start "start term" sets both sides of current equation to "term" and dependencies to null. It backs up the previous environment onto the desktop and sets the "current theorem" environnment variable to the null string (walk command "s").

startfor "startfor thm term" acts as "start", except that it sets the "current theorem" variable to "thm".

starthere "starthere" backs up the environment, then sets both sides of the current equation to the current whole term and dependencies to null.

startover "startover" backs up the environment, wipes dependencies, and sets both sides of the current equation to its left side (usually the term one started with). (walk command "B").

workback "workback" interchanges the left and right sides of the current equation (so one will be looking at the original term entered) (walk command "b").

getleftside see editing commands

getrightside see editing commands

autoedit see editing commands

11.8.4 Environment Display Commands

look View the whole term (above) and the current subterm (below) (walk command "?").

lookback View the left side of the current equation.

lookhere View the current subterm alone.

lookhyp "lookhyp integer" displays the hypothesis indexed by "integer"; see the discussion of the hypothesis function of the prover above.

lookhyps Displays all hypotheses.

seedeps See the dependencies of the theorem under construction. (walk command "j").

11.8.5 Display Control Commands

declareprefix "declareprefix infix default" causes the infix "infix" to be usable as a unary prefix, with invisible left subterm "default" (walk command "X").

deprefix "deprefix infix" removes any default prefix assigned to "infix".

setprecedence "setprecedence infix number" causes the infix "infix" to be assigned the precedence "number" (this parameter is not enclosed in double quotes). Non-negative integers are permitted as precedences. Odd precedences associate to the left; even precedences to the right. Precedence information is saved in saved theories; however, theorems are recorded in theory files in the old syntax. The default precedence for any infix is 0 (walk command "I").

showprecs Displays precedence information in a tabular format; also gives left or right association information. (walk command "k")

clearprecs Sets all precedences back to 0 (so we are back to the original precedence convention of the prover; no precedence and all operations associate to the right).

setline "setline number" sets the point at which lines wrap around in the display to "number" (the argument is *not* enclosed in double quotes). (walk command "K")

setdepth Sets the depth of the display (terms below this depth will not be seen—they are replaced with some impossible string). The form of the command is "setdepth number", and the number is not enclosed in double quotes. If the number is negative (as it usually is), all subterms will be displayed.

nodepth Sets depth to a negative value so that all subterms are displayed.

declaretypeinfo `declaretypeinfo variable type` enables the term `:?variable` (the variable parameter is currently entered without leading `?`, though this will probably be changed) to be read as `type : ?variable`; this allows implicit typing of variables. This information is not currently saved in theories.

detypeinfo `detypeinfo variable` turns off any implicit typing of the variable `?variable`.

11.8.6 Movement Commands

right Changes current subterm to right subterm of previous current subterm. Takes the immediate proper subterm of a constant function term (walk command "r").

left Changes current subterm to left subterm of previous current subterm. Takes the immediate proper subterm of a constant function term (walk command "l").

up Changes current subterm to the smallest subterm properly including the previous current subterm (walk command "u").

top Changes current subterm to the whole term (walk command "t").

upto "upto term" repeats "up" until a term is encountered which matches "term" in structure, or until the whole term is reached. (walk command "o")

downtoleft "downtoleft term" goes to the leftmost term matching "term" in structure. (walk command "<")

downtoright As "downtoleft", except rightmost. (walk command ">")

New movement commands: Commands `uptols <theorem>`, `uptors <theorem>`, `dlls <theorem>`, `dlrs <theorem>`, `drrs <theorem>`, and `dlrs <theorem>` enable movement to places where theorems can be applied. Commands beginning with `upto-` implement "upto"; commands beginning with `dr-` and `dl-` implement "downtoleft" and "downtoright"; the suffix indicates "left side" or "right side" of theorem.

11.8.7 Declaration Commands

showdec "showdec term" displays the declaration information for the constant or infix "term". (relative types of left and right subterms and default left subterm if used as a prefix). (walk command "~"). Displays comments.

showalldecs Shows all declarations, one per keystroke on Enter. Use any key other than Enter to break out.

declaretypedinfix "declaretypedinfix left-type right-type name" declares infix "name" with left and right types "left-type" and "right-type" (the types are integers without double quotes; negative integers are prefixed with ~ in ML). (walk command "Y").

declareconstant "declareconstant name" declares a constant with name "name". (walk command "c").

declareinfix "declareinfix name" declares a type-level infix (both relative types 0) called "name". (walk command "d").

declareunary "declareunary name" declares a type-level infix called "name" usable as a prefix with default left subterm `defaultprefix`. (walk command "1").

newcomment "newcomment name comment" attaches the comment "comment" to the declared object named by "name". This supersedes older comments.

appendcomment As newcomment, except that comments are appended to previous comments.

makeunary "makeunary name" assigns the default left argument "defaultprefix" to an infix which may already exist (it does not attempt a declaration).

declareopaque "declareopaque infix" causes the infix "infix" to be treated as referentially opaque for purposes of abstraction. (walk command "y") New restriction: this command can be used only to introduce a new infix; previously existing operators cannot be made opaque.

makescin, makescout "makescin name theorem" causes the function constant or infix named by "name" to be declared as having input (resp. output) restricted to absolute types (strongly Cantorian sets) with consequent weakening of stratification requirements. The theorem named by "theorem" must witness this fact.

declarepretheorem "declarepretheorem thm" reserves "thm" as the name of a future theorem (needed for the proofs of recursive tactics). (walk command "P").

forget "forget name" eliminates declarations of all theorems which depend on or refer to theorem "name". It will eliminate an entire module if any component depends on "name".

11.8.8 Theorem Display Commands

thmdisplay "thmdisplay thm" causes theorem "thm" to be displayed. (walk command "D"). Displays comments. Displays names but not contents of module components (hidden theorems).

showalltheorems Causes a theorem to be displayed with each keystroke (most recently proved first). Any key other than Enter breaks out. (walk command "T" has a similar function; use "l" and "r" to scan through the theorem list after issuing the "T" command)

moddisplay, showallmodules As thmdisplay, showalltheorems, except that module contents are displayed.

seeprogram Displays the theorem (if any) bound to a constant or infix.

seeallprograms Displays all programs.

seedef "seedef defined" displays the defining theorem of the constant or operator "defined".

seealldefs Displays all definitions.

showrelevantthms Displays theorems applicable to the current subterm. Will not display theorems which apply to *all* terms. Hitting any key other than Enter will break out.

showallaxioms Displays all axioms and definitions.

11.8.9 Proof Commands

axiom "axiom name left-side right-side" causes an axiom with name "name", left side "left-side" and right side "right-side" to be created.

interaxiom If the current subterm is an equation, "interaxiom axiom" introduces it as an axiom called "axiom". (walk command "x").

makeanaxiom "makeanaxiom thm" makes theorem "thm" an axiom.

proveanaxiom If the current equation proves an axiom "axiom" (verbatim) from other axioms, the command "proveanaxiom axiom" will reset dependencies globally to reflect this fact.

prove "prove thm" stores the current equation as "thm". "thm" may have variable parameters. (walk command "p").

autoprove Stores the current equation as a theorem with name the "current theorem" environment variable.

reprove See editing commands

autoreprove See editing commands

redefinition commands See the newer Command Reference.

module system commands See the newer Command Reference.

11.8.10 Editing Commands

applythm "applythm thm" applies the theorem "thm", but without executing any embedded theorems introduced. "thm" may be supplied with parameters.

applyconvthm As "applythm", but in the reverse direction.

applyenv, applyconvenv As the previous two commands, but using saved environments.

reprove "reprove thm" allows the current equation to be stored as "thm", where "thm" already exists, with the sole constraint being that the new version of "thm" must depend on no more axioms than the old version of "thm".

autoreprove Same effect as "reprove", with the same restriction, with the argument set to the "current theorem" environment variable.

getleftside "getleftside thm" gets left side of theorem as both sides of current equation and sets "current theorem" to "thm". Saves old environment on desktop. (walk command "L").

getrightside As "getleftside", but the right side of the theorem. (walk command "R").

autoedit "autoedit theorem" sets the current equation to "theorem", and sets the current theorem environment variable to "theorem" (with parameters, if any). (saves old environment on desktop). (walk command "E").

11.8.11 Definition Commands

defineconstant "defineconstant left-side right-side", where left-side has the correct format (an undeclared constant possibly applied to lists of parameters) declares the new constant read from "left-side" and introduces a theorem with null dependencies of the form "left-side = right-side", with the same name as the constant defined.

defineinfix "defineinfix thm left-side right-side" where "left-side" has a top level undeclared infix (or prefix) linking and possibly applied to parameters or lists of parameters, declares the new infix (as type level) and creates a theorem called "thm" equating "left-side" and "right-side".

definetypedinfix "definetypedinfix thm left-type right-type left-side right-side" has the same effect as "defineinfix thm left-side right-side", except that the left and right sides of the new infix are assigned relative types "left-side" and "right-side": these arguments are numerical and are *not* put in double quotes.

defineitconstant Like "defineconstant", except that the right side is the current subterm.

defineitinfix Like "defineinfix", except that the right side is the current subterm.

defineittypedinfix Like "definetypedinfix", except that the right side is the current subterm.

defineopaque Defines opaque infixes. Same arguments as "definetypedinfix", except that types are not needed.

11.8.12 Rule Introduction Commands

ruleintro "ruleintro theorem" introduces an embedded occurrence of "theorem" at the current subterm. (walk command "i").

revruleintro As "ruleintro", except that the theorem is to be applied in the reverse direction, (walk command "v").

undelay Remove a "delay" prefix from the current subterm.

delay Introduce a "delay" prefix to the current subterm.

unlazy, lazy As "undelay" or "delay" with the full laziness prefix.

altrule Converts an embedded theorem at the current subterm between the normal and alternative forms. (walk command "f").

droprule Drops an embedded theorem from top of current subterm (if there is one).

proveprogram "proveprogram name thm" causes the theorem "thm" to be applied wherever the constant or infix "name" appears in a context defined by the left side of "thm". Simulates functional programming.

deprogram "deprogram name" removes the theorem bound to "name".

11.8.13 Assignment Commands

assign "assign variable term" replaces the variable "variable" with the term "term" throughout the equation under construction. (walk command "a").

assigninto "assigninto variable term" replaces the variable "variable" with the two sides of the equation under construction to produce a new equation under construction. (walk command "n").

assignright "assignright var thm" assigns the right side of theorem "thm" as the value of the variable "var" throughout the current and initial terms of the theorem under construction.

assignleft As "assignright", but the left side of "thm".

11.8.14 Tactic Language Interpreter

steps The trace/debug command for the tactic language; executes one step in current subterm each time a key is pressed. Hit "q" to break out; terminates automatically if the term stabilizes.

onestep More limited trace/debug command; executes just one step in current subterm.

execute The tactic language interpreter; executes all embedded theorems or tactics in the current subterm, and all embedded theorems or tactics newly invoked in this way, using an aggressive depth-first strategy. (walk command "e").

Warning (for all these commands): If you break out of any tactic interpreter using Control-C, use the "top" command to reset environment variables.

11.8.15 Theory Desktop Commands

scantheories Allows one to scan the theories on the desktop; hit "h" in any "scan" command to see the local commands to move from item to item or to quit.

cleartheories Erases all theories on the desktop.

droptheory "droptheory theory" drops theory "theory" from the desktop.

gettheory "gettheory theory" loads the theory "theory" from the desktop (resetting the "current theory" environment variable). It automatically backs up the current theory unless the current theory is the one being loaded.

backuptheory Saves current theory on desktop.

11.8.16 Environment Desktop Commands

getenv "getenv env" allows one to get the environment (equation under construction in the current theory) called "env". The current environment is saved unless it is called "env" itself.

saveenv "saveenv env" allows one to save the current equation as "env".

backupenv Saves the current environment using the "current theory" environment variable as name; if the current theory is null, calls it "backup".

dropenv "dropenv env" erases environment called "env".

clearenvs Erases all environments on the desktop.

loadsavedenvs Loads save environments in the theory onto the desktop; one must hit return for each theory. One can use this in combination with clearenvs to unclutter theory files.

scanenvs Allows one to scan the environments on the desktop (in the current theory). Type "h" after typing any "scan" command to see the local commands to move from item to item or to quit.

11.8.17 View Management Commands

showviews This command shows all views in a tabular format.

showview "showview view" shows the view "view" in a tabular format (translation of terms in the source theory to terms in the target theory; views are stored as part of the source theory).

viewofin "viewofin ident view" shows the translation of "ident" in the view "view".

clearviews Erases all views.

dropaview "dropaview view" erases the view "view".

dropfromview "dropfromview view ident" drops the translation of "ident" from the view "view".

viewasselfin "viewasselfin ident view" causes "ident" to be translated by "ident" in view "view".

viewasin "viewasin name1 name2 view" causes "name1" to be translated as "name2" in view "view" (name1 must be an identifier in the current theory).

declareview "declareview view" creates a view with some default content.

11.8.18 Theorem Export Commands

exportthm "exportthm view prefix theorem remote-theory" exports "theorem" and all iteratively embedded theorems to "remote-theory", using "view" to set up the translation, and adding "prefix" to each theorem name. If the view does not check, no changes will be made. "#" is used as the default prefix for infixes which are created during the export. "remote-theory" must be on the desktop.

exportthmlist "exportthmlist view prefix1 prefix2 theorems remote-theory" is a more general version of exportthm above: the differences are that "theorems" is a list of theorems to be exported, and that prefix1 (for theorem and constant names) is supplemented with prefix2 (for infix names).

11.8.19 Command Abbreviations

```
(* /////////// ml declarations to change the names of commands //////////// *)
fun ri x = ruleintro x;
fun rri x = revruleintro x;
fun ari x = altruleintro x;
fun arri x = altrevruleintro x;
fun s x = start x;
fun p x = prove x;
fun rp x = reprove x;
fun ex() = execute();
fun di x = declareinfix x;
fun du x = declareunary x;
fun a x = axiom x;
fun dpt x = declarepretheorem x;
fun sp x n = setprecedence x n;
fun td x = thmdisplay x;
fun wb() = workback();
fun ae x = autoedit x;
fun tri x = targetruleintro x;
fun smt x = showmatchthm x;
fun srt() = showrelevantthms();
fun dti s t = declaretypeinfo s t;
fun uti s = detypeinfo s;
fun sat()  = showalltheorems();
```